Code Review Scope for Integrity Breaches by Insider

By

Pushkar Ogale B.E., M.C.S.

A DISSERTATION

In

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University, in
Partial Fulfillment of
the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Approved

Dr. Michael Shin

Committee Chairman

Dr. Yu Zhuang

Dr. Sunho Lim

Dr. Fang Jin

Accepted

Dr. Mark Sheridan

Dean of the Graduate School

May, 2017

ACKNOWLEDGEMENTS

I would like to convey my sincere appreciation and a note of thanks to my advisor Dr. Michael Eonsuk Shin for his continuous guidance, advice and encouragement during the past 5 years. It has been a great experience to be Dr. Shin's Ph.D. student. I am truly grateful for all the time he invested in helping me achieve my goals. He was available at all times to provide wonderful advice when the going got difficult and there seemed no easy solution. I acknowledge Dr. Shin's advice and deep knowledge and the many insightful suggestions that have helped me immensely in progressing my research and developing my research skills through valuable feedback.

I am immensely grateful to my Ph.D. committee members, Dr. Yu Zhuang, Dr. Sunho Lim and Dr. Fang Jin for the valuable feedback and encouragement I have received from them. I want to thank the entire committee for providing the guidance on the outcomes to focus on as a part of my dissertation.

I would also like to deeply appreciate Dr. Hewitt Rattikorn, Chairperson of Computer Science Department, and Dr. Audra Morse, Professor and Associate Dean of Undergraduate Studies, College of Engineering, who have provided me with the support to be able to focus on and complete the requirements of my Ph.D. Dr. Hewett was instrumental in providing me initial support upon arriving at the University and Dr. Morse was instrumental in providing continuing support while I pursued the degree. Without their support and assistance, this Ph.D. would not have been possible. Together both of them enabled me to pursue my dream of being a teacher. I am happy to say that I have been able to teach full instruction of courses at Texas Tech University.

I also acknowledge the excellent Faculty at Texas Tech University for the part they played in my learning. The outstanding coursework coupled with the challenging projects have helped in developing skills essential to an academic and research career. I feel honored to have been presented an opportunity to be a student at the Department of Computer Science at Texas Tech University.

Beyond the academic area, I want to deeply acknowledge my family members Vaishali, Arjun and Anisha for showing remarkable tolerance to having another student in their family. They showed immense support in me achieving a long-time dream of accomplishing a Ph.D. in my field. They have patiently endured hardships without any complaints. Without their support, this achievement would not be possible.

I also acknowledge my parents Vishwanath and Shyamala for their ever-present support.

I would like to acknowledge the staff of the Computer Science Department for their unrelenting support to all students in executing their degree plan efficiently and effectively toward completion of the degrees.

A special thank you to Dr. Susan Mengel, Dr. Susan Urban, Dr. Noe Lopez, Dr. Akbar Namin, Dr. Eunseog Youn, Dr. Richard Watson and Dr. Yu Zhuang who were my Professors while at the University. I learnt a lot from being their students.

Finally, I would like to acknowledge all of my team mates in projects in the coursework I completed. I have learnt a lot from my team mates and it has been a great experience working with ethnically diverse team mates each of whom brought in their own specialties to the team.

TABLE OF CONTENTS

ABSTRACT

Integrity security is a key element of secure applications. To ensure that application has integrity security, the process to follow is to review the source code that is being developed by the software engineers.

The code review process is a tedious and time consuming process, the complexity of which is directly dependent on the size of the application being developed. Due to the nature of the code review process, many times the reviews may not be conducted as thoroughly as required. There is also an aspect of the skill level of the reviewer that may be inadequate for the level of expertise required to review and provide feedback or corrections for a secure application.

There is a need to identify the scope of mandatory code review, which are parts of the source code that should absolutely be manually scrutinized and reviewed for ensuring that the application satisfies the integrity security requirements. Insider malicious attacks are carried out by insider software engineers either for personal gains or to cause damage to their employers. One way to cause damage is to write malicious source code that causes either immediate damage or time lapse damage. There is a need to address insider attacks that are caused by malicious code.

This dissertation introduces an approach that allows us to determine the scope of code review for secure applications, the integrity of which can be compromised by an insider with malicious intent. The approach identifies

suspicious codes that might be contaminated by an insider in object-oriented programs so that it draws the reviewer's attention to these codes. The goal of the proposed approach is to mitigate the code review for the integrity security of a program by providing the scope of code review instead of reviewing the whole program for an application. The integrity breach conditions (IBCs) are specified using the concepts of coupling in a program and the conditions are used to find security spots that might contain malicious codes. IBCs are specified using Object Constraint Language (OCL) with a meta-model for Java.

A Code Analysis for Integrity Security (CAIS) tool is developed and provided to validate the approach proposed in this dissertation. The tool can be used to verify object oriented secure applications written in the Java programming language.

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

1. INTRODUCTION

Insider attacks are carried out covertly to compromise the security of critical applications, so it is difficult to detect the attacks. Insiders [Collins13, Silowash12, Shin10, Shin11], such as software engineers or third-party contractors, may intend to earn personal gain from applications or disrupt applications due to the discontent with organizations. Insiders can add malicious codes to an application in software development or change existing codes maliciously in maintenance such that those codes are activated either periodically or non-periodically to earn gain, disrupt services or destroy applications and/or data. However, it is very difficult to detect insider attacks immediately because the attacks are committed covertly and the harm they cause is revealed slowly. Most of the organizations become aware of the attacks after they have suffered damage.

Code review is one of the approaches that can be used to detect security breaches caused by insiders who participate in developing or maintaining the code of applications. Code review is a systematic approach to examine the programmer's mistakes in the code so that it improves the reliability of applications. Code review can identify the insecure piece of code for applications, which may lead to the vulnerabilities of applications, as well as find the mistakes overlooked in the software development. The *Insider Threat Study* [CERT13] suggests that the organizations developing and maintaining secure applications must consider insider threats by means of appropriate approaches such as code review [Silowash12].

Although code review for applications can detect malicious codes, all the program codes in a large application cannot be practically reviewed line-by-line under the pressure of tight project deadline. Code review in an organization is a cumbersome and time-consuming process. Many times, the code review process is not carried out in the true spirit of the development and maintenance process. Also, code review may not be implemented due to the lack of resources whenever applications change or evolve according to changes in business logics. In addition, code review for identifying insecure code may not be carried out systematically in an organization because it relies on the security expertise and experience of reviewers. In practice, code review for security should be applied to just part of codes in an application, referred to as a **security spot**, which might contain a malicious code in terms of the integrity security of the application.

This dissertation aims at developing an approach to determining the security spot in an application for code review against insider attacks. This dissertation focuses on the **integrity** security goal of an application in that the security assets, such as account balance or password, should not be changed by insider attacks. Malicious insiders may add new codes to the application or change existing codes in the application so that the security assets of the application are compromised. This dissertation focuses on identifying the security spot of application that might be changed maliciously or to which new malicious code might be added by insiders.

This dissertation describes an approach that determines the scope of mandatory code review against the breach of integrity security that might be committed by a malicious insider. The approach points out the security spots in a program, each of which might be possibly exploited by a malicious insider to compromise the program. The approach

specifies the integrity breach conditions (IBCs) using the concepts of coupling between objects in the object-oriented programming. The approach adopts the concepts of content, external, data/stamp, subclass, and package couplings to find the IBCs.

CHAPTER 2

2. RELATED WORK

Several approaches have been proposed to analyze security vulnerability concerns in programs for addressing confidentiality, integrity, privacy and availability of applications. Multiple techniques such as secure information flow analysis [Smith06], static/dynamic taint check analysis [Jovanovic06, Newsome04], string analysis [Yu14], secure program execution [Lee04], capture of system wide information flow [Yin07], and discovering attacks using symbolic execution [Pattabiraman09] have been proposed for these security concerns.

Secure information flow analysis [Smith06] involves performing static analysis of a program with the aim of proving that no sensitive information is leaked. This type of analysis requires a flow policy to be established and enforced for every program, especially one that deals with secure information, so that there is upfront agreement on how much information can be shared and at what levels. The necessary analysis capability is built so that the flow of information can be monitored and the policy is enforced. The program can be executed safely if it passes the analysis. This type of analysis is aimed primarily at the confidentiality property of secure information.

In static taint check analysis [Jovanovic06], the problem of vulnerable Web applications is addressed by static source code analysis. Flow-sensitive, inter-procedural and context-sensitive data flow analysis is employed to discover vulnerable points in a program. This approach is targeted at the general class of taint-style vulnerabilities. The concept behind taint checking is that any variable within a program that can be modified by an outside

user (for example a variable set by a field in a web form) poses a security risk to the program. If that variable is used in setting values of a second variable, then that second variable is now also suspicious. The taint checking tool checks variable by variable to build a list of all variables that are potentially affected by the outside input. If any of these variables is used to execute dangerous commands (such as commands to a SQL database), the taint checker warns that the program is using a potentially dangerous tainted variable. This concept is applied to the detection of vulnerability types such as SQL injection, cross-site scripting, or command injection. This type of analysis is aimed at securing the availability of applications, and integrity, privacy and confidentiality of information.

Dynamic taint check analysis [Newsome04] is proposed for automatic detection and analysis of overwrite attacks, which include most types of exploits. The analysis is done at runtime as opposed to static analysis. Tainted input data from untrusted sources is monitored at runtime to track how the predefined tainted input propagates and to check when tainted data is used in dangerous ways. The analysis also identifies important invariants of the tainted data that can be used as signatures to detect attacks. This approach does not need any source code or special compilation techniques for the program being monitored and thus works well on commercial of the shelf (COTS) software.

String analysis [Yu14] is used on a regular expression pattern or string within a program to describe the potential contents of every string variable at any point in a program. An important observation made by the authors in [Yu14] is that most vulnerability is caused by improper string manipulation. Programs that use or propagate malicious user inputs

without proper sanitization are vulnerable to attacks. This analysis determines all possible dangerous string constructs that can be built with the current string composition and provides a warning when the contents of the string could result in an exploit.

Secure program execution [Lee04] addresses the transfer of a program's control to malevolent code when a security attack takes control of a program. In this approach the operating system would identify a set of input channels as spurious, and it will track all information flows from the identified inputs. A broad range of attacks are effectively defeated by disallowing the spurious data to be used as instructions or jump target addresses. This approach aims to address the confidentiality and the availability of software.

In capture of system wide information flow, the authors in [Yin07] observe that malicious information access and processing behavior is the fundamental trait of numerous malware categories, which breach user's privacy (including key loggers, password thieves, network sniffers, stealth backdoors, spyware and rootkits). The authors in [Yin07] propose a system, Panorama, which detects and analyzes malware by capturing this fundamental trait in most malwares and separates these malwares from benign software. This technique aims to address the confidentiality or privacy aspects of software. The authors propose a fine-grained taint tracking for the whole system to monitor taint propagation after introducing a tainted sensitive code into the system. Taint graphs are generated through the analysis. The authors define various policies based on these taint graphs to specify the behavior of different malware. By checking the policies against the taint graph of an unknown code, the approach enables the automatic detection and analysis of malicious code.

In using symbolic execution for discovering application level insider attacks [Pattabiraman09], the authors suggest that the developer has the best knowledge of the locations in the program where an insider can launch an attack. These attack points are identified and the program is executed with a known input. When the execution reaches the specified attack point, then a single variable is chosen from the set of all variables in the program and a symbolic value is assigned to that variable instead of a concrete value. The other variables in the program are not touched. This procedure is repeated exhaustively for each data value in the program at each of these attack points. This simulates all the insider attacks on a given application program

Most of the current approaches focus on security breaches caused by insecure input, string or spurious channels against confidentiality, privacy, and availability. However, our approach proposed in this dissertation addresses the code review for security spots against the breaches of integrity security that might be compromised by insider attacks. Our approach is different from taint analysis or string analysis that deals with the same integrity security. The figure below compares the different approaches with our approach.

Approach	**What artifacts these work on**	**What are the Security Goals**	**Vulnerability detected by this approach**	**Detect Insider attacks**	**Methodology**
Secure Information Flow	Source Code/ Secure information residing in variables in a program	Confidentiality : Ensure that no confidential, secure information is leaked.	Leakage of information from program	NO	**1.**Static analysis **2.**Prevent leakage of secure information

Static Taint Analysis	Source Code User Input	Confidentiality / Integrity/ Availability	Malicious code execution at runtime	NO	**1.**Analyze flow of tainted information **2.**Prevent malicious code from executing
Dynamic Taint Analysis	Source Code, Executables / Binaries User Input	Confidentiality / Integrity/ Availability	Detection of flow through program that caused an attack	NO	**1.**Analyze binaries for vulnerabilities. **2.**Trace back from detection point **3.**Identify the flow causing an attack.
String Analysis	Source Code/ Input String/ String variables	Integrity/ Availability/ Access Control	Ensure that malicious commands cannot form using string variables	NO	**1.**Analyze all string variables **2.**Prevent formation of malicious commands
Secure Program Execution	Executables / binaries in specific channels of Operating System	Confidentiality and Availability. Ensure that no malware can take hold of the system	Operating system monitors attacks from channels marked as spurious	NO	**1.**Identify set of spurious input channels. **2.**Track information flows from these inputs. **3.**Disallow spurious data to be used as instructions or jump target addresses.
Capture of system wide information flow	Source Code/Input String/ String variables	Confidentiality / Privacy	Fine grained taint tracking to enable automatic detection of	NO	**1.**Fine grained taint tracking for system **2.**Introduce and monitor taint

			malicious code at run time		propagation in system. **3.**Generate Taint graphs and define policies based on these graphs **4.**Check policies to enable the automatic detection and analysis of malicious code.
Our Approach (CAIS)	Source code	Integrity	Adding or changing code maliciously by insiders	**YES**	**1.**Scan source code to identify potential malicious code 2.Lead security spot to code review

Figure 1 Comparison among approaches

CHAPTER 3

3. PROPOSED APPROACH

3.1. Example of Integrity Breaches by Insider Attack

An insider attack can jeopardize the integrity security of an application by planting malicious code to the application. The malicious code might alter the behavior of the application in terms of security. The malicious code can be created in a program by adding a new code to or modifying the existing code in the program, running when it is called or a specific time is reached.

Fig. 1 depicts an example of malicious code that is added to an ATM application. The Account class encapsulates the account number and balance variables in which the balance of account should not be changed insecurely. The Account class provides the Account() operation to construct an object, and credit() and debit() operations to manipulate the balance of account. The original credit(double amount) operation has been designed with only one argument to increase the balance as much as the amount, but an insider can add a new malicious operation credit(double amount, Account accNumber) operation to the Account class so that he/she adds the amount to the balance of a malicious account using the operation. The main() function shows that this happens when the na.credit(30, ma) operation is called instead of na.credit(30). Fig. 1 also shows that the debit() operation in the Account class has been changed maliciously. The debit() operation has been originally designed just to subtract the amount from the balance of account, but an insider changes it not to decrease the amount from the balance. A malicious insider might want to do harm to the application due to his/her discontents.

```
public class Account {
	private int number;
	private double balance;
	public Account(int number) {
		this.number = number;
		balance = 0;
	}
	public void credit(double amount) {
		balance = balance + amount;
	}
	public void credit(double amount, Account accNumber) {
		accNumber.credit(amount);
	}
	public void debit(int amount) {
		balance = balance;
	}
	public double get() {
		return balance;
	}
	public static void main(String[] args) {
		Account na = new Account(1);
		Account ma = new Account(2);
		//na.credit(30);
		na.credit(30, ma);
		System.out.println("na balance = " + na.get());
		System.out.println("ma balance = " + ma.get());
	}
}
```

Figure 2 Malicious code added to ATM Application

In other case, a malicious bomb code can be planted into an application by an insider who may be convinced of her/his lay-off. After the insider leaves, the malicious (logic or timer) bomb code may be activated to repeatedly to delete the data that requires the integrity security for the application. For example, a malicious bomb code may debit some amount randomly from customer checking accounts so that it disrupts system services or destroys account data.

3.2. Overview of Approach

The proposed approach is to identify any security spot in a program that might compromise the integrity of program. While an object-oriented program runs, a secure data is stored in a database or maintained in a variable of an object. A malicious insider can tamper the secure data so that the program is compromised. Our approach focuses on the secure data for a program to find the security spots for code review.

Fig. 3 depicts the overview of the approach, which identifies security spots of a given program for code review. The approach analyzes the source code of a program by means of the inputs - a program code that is a source code for an application, the integrity breach conditions (IBCs) that specify criteria for finding security spots for code review, and the secure data in a program that should be protected from a malicious insider attack. The output of the approach is the scope of code review, which is presented with all the methods (operations) of objects in a program that require mandatory code review. Each method included in the scope of code review is referred to as a code review (CR) method (operation) in this dissertation, whereas a method that is not included in code review is called a non-code review (NCR) method.

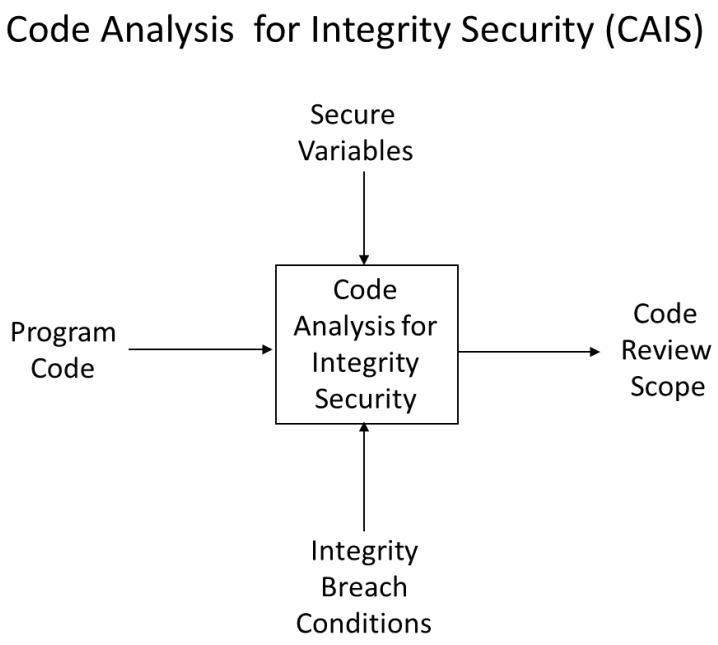


Figure 3 Our Approach

The secure data in a program can be stored in a variable in an object (class) or in a database such as a relational database. In this dissertation, a variable that stores a secure data is referred to as a secure variable (SV) whereas a variable that does not contain any secure data is called a non-secure variable (NSV). A secure class (SC) is defined as a class encapsulating at least one secure variable, while a non-secure class (NSC) does not contain any secure variables. For a database, we assume, for the simplicity of our approach, that database wrapper classes (DWCs) [Gomaa2000] are designed for a program to access the data in a database. A DWC is used for modularization and easy maintenance of a program. When an object accesses a data in the database, it needs to call the methods provided by the DWC. A database wrapper class does not have any variables (e.g., class or object variables), hiding the detailed logic to access the database. In this dissertation, a secure database wrapper class (SDWC) is defined as a DWC that has at least one method accessing a secure data stored in a database, while a non-secure

database wrapper class (NSDWC) is defined as a DWC that does not have any methods to access secure data in the database.

This approach defines the integrity breach condition that is the criteria to determine whether a code can be a security spot. A security spot can be used to change the value of a secure data in a program maliciously if an integrity breach condition holds. An integrity breach condition can also point out a legitimate code as a security spot, which is designed to change the value of a secure variable regally in a program. This is because the integrity breach conditions hold for all the code pieces that can change the values of secure variables.

The integrity breach conditions for code review decision can be developed two-fold – static conditions that are specified for the methods of classes in a program; and dynamic conditions that are defined for tracing the program execution at runtime. The integrity breach conditions used in the approach (Fig. 1) contain all the static and dynamic integrity breach conditions.

In order to identify and specify the integrity breach conditions, we need to be able to analyze a meta model of a language and use the Object constraint Language (OCL) to specify the conditions. The use of a meta-model allows us to ensure that all possible language syntax is being addressed. Use of OCL allows us to specify the conditions in terms of constraints that are them codified in an efficient manner.

CHAPTER 4

4. SCOPE OF CODE REVIEW FOR INTEGRITY

For finding the security spots requiring mandatory code review in a given program, we focus on the relationships between a secure data and a method (operation) in a class in which the method changes the secure data value. A method containing a malicious code can alter directly a secure data value or it can change a non-secure data value, which might be referenced by other methods to modify the secure data value. A method containing a malicious code might call another method through which it changes the secure data value insecurely. A secure data value in a class might be contaminated by a method in a subclass of the class. The malicious method (code) can be traced by means of the relationships between the secure data and methods in a program.

The relationships between a secure data and a method in a program are analyzed using the concepts of coupling [Pfleeger09]. In general, coupling shows the dependency relationships between two modules in a program. Coupling is the extent or degree that a module communicates or shares data/control with another module, and it is a factor that determines the quality of software design in software engineering. The couplings between modules occur in several ways, such as content, external, data/stamp, subclass or package couplings, each of which shows how a module affects another module. Using the concept of coupling, we evaluate how a method can change a secure data value.

Coupling in object-oriented programming takes place between objects (or classes or components) when an object passes a data to another object that processes the data. An object dealing with a secure data can pass the secure data to another object via coupling.

It is possible that the sharing of secure data between objects could result in inadvertent or malicious use of the data, which may corrupt or destroy the integrity of the secure data. It is necessary to review possible couplings in an object-oriented program so that we identify the code that might be used to tamper or destroy the integrity of secure data.

In this dissertation, all the methods in a program are NCR methods at the beginning of analysis of a program. As our approach evaluates each NCR method in a program, the status of a NCR method changes to a CR method if the NCR method satisfies an integrity breach condition. For example, a NCR method becomes a CR method if it turns out to be changing the value of a secure variable directly, or affecting other CR methods indirectly. Only CR methods are included in the scope of mandatory code review.

Before listing all the IBCs, we ensure that all variables in the classes are declared as private or protected.

IBC (Class Variable declaration): *A variable V is defined in a class, and it is not defined as private or protected, then flag the declaration.*

4.1. Code Review Scope with Content Coupling

Content coupling occurs when a data of a module is directly accessed by another module to read or modify the data. Since a module depends on the inner workings of another module in content coupling, a secure data in a module might be changed by another module. When this happens, the integrity of a secure data in a module might be no longer assured.

In object-oriented programming, any object can gain access to a variable of another object if the variable is declared as public. A variable declared in a class has its access modifier, such as public, private, and protected in object-oriented programming, which

constraints the access of other objects to the variable. A variable with the public modifier allows all objects to gain access to the variable, whereas the private modifier confines the access within a class. For the protected modifier, a child class can access the variables of its parent class. In general, a variable declared as private allows other objects to access it via only its setter or getter method. A variable in a class is at risk of being maliciously changed by accessing the variable directly if it is declared as public. We assume in this dissertation that all variables of classes are declared as private or protected, but not public. Also, we assume that a variable declared as protected in a class can be only accessed by the child classes of a parent class, but not by a class outside of the inherited relationship. This is to prevent a secure data from being tampered by accessing it directly.

In this dissertation, content coupling is used in a narrow sense for the dependency between a variable (e.g., a class or object variable) and a method within the same class. Although content coupling addresses the dependency relationship between classes (objects) in the object-oriented programming, a class in our approach cannot access directly a variable of another class to modify because we assume that all the variables defined in a class are private or protected. However, a method in a secure class can directly modify the value of a secure, private variable in the class or protected variable declared in its parent class. This is not exactly content coupling between classes, but the concept of content coupling is adopted in our approach to apply it to the dependency relationship between a variable and a method within the same class. On the other hand, a class can modify indirectly the value of a variable defined in another class by calling a setter method provided by the called class. Such a dependency relationship between

classes will be handled in the data/stamp coupling in this dissertation, separately from content coupling.

Associated with the concept of content coupling, there are several cases that a method in a secure class can modify the value of a secure variable in the class. The following describes such cases with their integrity breach conditions.

4.1.1. Direct Change in Secure Class

A method defined in a secure class can directly access a secure variable in the class to change the value of the secure variable. A malicious method (e.g., credit(double, Account) method in Fig. 1) can be added to a secure class to compromise the value of a secure variable (e.g., balance variable in Fig. 1). The following describes the integrity breach condition (IBC) for *direct change in secure class*. A method M in the following IBC can be a CR method if the IBC is true:

IBC (Direct Change in Secure Class): *A method M is defined in a secure class, and it changes the value of a secure variable in the class.*

4.1.2. Change to Variable Referenced by CR Method in Secure Class

A method defined in a secure class can change the value of a non-secure variable in the class, which is referenced by a CR method in the class. In the case, the method that changes the value of a non-secure variable comes to be a CR method. The IBC for *change to variable referenced by CR method in secure class* is to detect a malicious code in a secure class that affects the value of a secure variable by changing the value of a non-

secure variable in the class. A method M in the following IBC can be a CR method if the IBC holds:

IBC (Change to Variable Referenced by CR Method in Secure Class): *A method M is defined in a secure class, and it changes the value of a non-secure variable in the class that is referenced by a CR method in the class to modify the value of a secure variable.*

The IBC for *change to variable referenced by CR method in Secure Class* can be extended to the case where a method M changes the value of a non-secure variable, which is referenced by a NCR method to change the value of another non-secure variable, which is referenced by a CR method to change the value of a secure variable. More than two non-secure variables can be involved in a change to the value of a secure variable.

4.1.3. Call CR Method in Secure Class

A NCR method in a secure class can call a CR method in the class to change the value of a secure variable though the CR method. A malicious code that has been added to a secure class can change the value of a secure variable by calling a CR method in the class. For detecting such a malicious code in a secure class, code review must include all NCR methods that change the value of a secure variable by calling any CR method in the class. The following is the IBC for *Call CR method in Secure Class* in which a method M can be a CR method if the IBC is true:

IBC (Call CR Method in Secure Class): *A method M is defined in a secure class, and it calls a CR method in the class to change the value of a secure variable.*

The IBC for *Call CR Method in Secure Class* can be extended to the case where a NCR method calls another NCR method, which calls a CR method to change the value of a secure variable. More generally, within a secure class, a NCR method can call a CR method through multiple NCR methods to change the value of a secure variable.

4.1.4. Method Called by CR Method in Secure Class

A method in a class can call any methods in the class. A malicious insider can compromise a NCR method in a secure class, which is called by a CR method in the class, so that the value of a secure variable is tampered by the NCR method. The compromised NCR method should be included in mandatory code review. A method M in the following IBC can be a CR method if the IBC holds:

IBC (Method Called by CR Method in Secure Class): *A method M is defined in a secure class, and it is called by a CR method.*

4.1.5. Call CR Method in Inner Class of Secure Class

In an object-oriented program, a class can contain its inner class in which a method of the inner class can change the value of a variable of its outer class. This is because the inner class can directly access any variables defined in the outer class. A malicious code can be added to an inner class so that it changes the value of a secure variable of the secure outer class. Also, an insider can add a malicious code to an inner class contained in another inner class because an inner class can encapsulate another inner class. The IBCs for inner classes can be specified to find the security spots associated with inner classes.

All inner classes should be analyzed to determine the scope of code review. However, we assume for simplicity that all inner classes encapsulated in a secure class are included in

mandatory code review. That means that all the methods in the inner classes of a secure outer class are CR methods.

A method in the outer class can call a method in its inner class in the object-oriented programming. Using this language feature an insider can add a malicious code to a method in the outer class, which calls a method in its inner class that is a CR method. Although all the methods in the inner classes of a secure outer class are reviewed as CR methods, the malicious code added in a method in a secure outer class tampers the value of a secure variable defined in the secure outer class by calling a method in an inner class of the secure outer class. The following IBC is necessary for detecting a method M that might be contaminated by such a malicious code:

IBC (Method in Inner class of Secure Class): *A method M is defined in an inner class of a secure class.*

IBC (Call CR Method in Inner Class of Secure Class): *A method M is defined in a secure outer class, and it calls a method in an inner class of the secure outer class.*

A method that contains malicious code in a secure outer class might call a method in the same secure outer class, which accesses a method in an inner class to tamper the value of a secure variable in the secure outer class. However, in this case, the method with the malicious code is detected by the IBC for *Call CR Method in Secure Class* as the method called by the malicious code in the secure outer class turns out to be a CR method.

4.1.6. Change to Variable Referred by Inner Class of Secure Class

A method in a class can access any variables defined in the class to change the values of variables. An insider can compromise a method in a secure outer class to tamper the value of a variable in the class, which might be referred by a method in an inner class of

the secure outer class (which updates the value of a secure variable in the secure outer class.) The method in an inner class is assumed to be a CR method in this paper and it can refer to the value of the variable in the secure outer class before changing the value of a secure variable in the secure outer class. The compromised method in a secure outer class must be included in mandatory code review. A method M in the following IBC must be a CR method if the IBC holds:

IBC (Change to Variable Referred by Inner Class of Secure Class): *A method M is defined in a secure outer class, and it changes the value of a variable in the class that is referred by a method in an inner class of the secure outer class.*

This IBC might be replaced with the IBC for Change to Variable Referred by CR Method in Secure Class (6.1B), but it ensures that the IBC (6.1B) needs to be applied to the non-secure variable accessed by a method in the inner class of a secure class.

4.1.7. Method Called by Inner Class of Secure Class

A method in an inner class can call any methods in a secure outer class. A malicious insider can add a malicious code to a method in a secure outer class that is called by a CR method in an inner class of the secure class. The compromised method must be reviewed. A method M in the following must be a CR method if the IBC holds:

IBC (Method Called by Inner Class of Secure Class): *A method M is defined in a secure outer class, and it is called by a CR method in an inner class of the secure outer class.*

This IBC might be replaced by the IBC for Method Called by CR Method in Secure Class (6.1D), but it ensures that the IBC (6.1D) needs to be applied to the method in an outer secure class, which accessed by a CR method in the inner class of a secure class.

4.2. Code Review Scope with External Coupling

External coupling occurs when a module is bound to another module via external devices, such as a database or files. For this dissertation, only external coupling through database will be considered. One module generates a data and stores it to an external device, whereas the other module reads the data from the external device to process the data. Once a module changes a data value in an external device, the effect of the change spreads to all the modules that access the data in the external device.

An assumption for the external coupling is that no Inner classes or constructors will be declared in classes that are Database Wrapper classes (DWC)

4.2.1. Direct Change to Data in SDWC

When a program stores a secure data in a database, a malicious insider can breach the security of the data by adding malicious code to the program. For detecting the malicious code, a method in a SDWC must be included in code review if the method is updating the value of the secure data. The method might be a target to which an insider wants to add a malicious code. The following IBC is defined to identify a method M that changes the secure data value in a SDWC if the IBC holds:

IBC (Change to Secure Data in DWC): *A method M is defined in a DWC and it changes a secure data value stored in a database.*

4.2.2. Call CR Method in DWC

A method in a DWC can call a CR method in the class, which might change or affect the secure data value in the database. A malicious insider can add a malicious code to a NCR

method in a DWC so that the NCR method compromises a secure data value in the database through a CR method in the class. A method M in the following IBC can be a CR method if the IBC holds:

IBC (Call CR Method in DWC): *A method is defined in a DWC and it calls a CR method in the class.*

4.2.3. Method Called by CR Method in DWC

A CR method in a DWC can call a NCR method in the class, which might affect the secure data value in the database that is updated by the CR method. A malicious code can be added to a NCR method in a DWC to compromise a secure data value. A method M in the following IBC can be a CR method if the IBC holds:

IBC (Method Called by CR Method in DWC): *A method M is defined in a DWC and it is called by a CR method in the class.*

4.3. Code Review Scope with Data/Stamp Coupling

Data or stamp coupling occurs when a module sends a data to another module. A module in data coupling sends a data to another module that processes the data, whereas a module in stamp coupling sends a data structure to another module that processes part or all of the values in the data structure. In data or stamp coupling, a sender module is bound to a receiver module via communication between the modules and the data shared by the modules.

In object-oriented programming, data or stamp coupling can be presented the interaction between the objects of classes. An object can call or be called by another object to transit a data using the methods of the objects. A malicious code can be added to a method in a class (note that this class can be any SC, NSC, SDWC, or NSDWC), which calls or is called by a CR method in another class (note that this also can be any SC, NSC, SDWC, or NSDWC). As a method in a class can become a CR method by means of data or stamp coupling, it is necessary for additional analysis to identify new CR methods in the class.

4.3.1. Call CR Method in other class

When a method in a class calls a CR method in another class, the method can tamper the secure data via the CR method. A method M in the following IBC can be a CR method if the IBC holds:

IBC (Call CR Method in other class): *A method M is defined in a class and it calls a CR method defined in another class.*

4.3.2. Method Called by CR Method in other class

A method in a class can be called by a CR method in another class. A malicious code can be added in the method in a class that is called by a CR method in another class. A method M in the following IBC can be a CR method if the IBC holds:

IBC (Method Called by CR Method in other class): *A method M is defined in a class and it is called by a CR method in another class.*

4.3.3. Change to Variable Referenced by CR Method in NSC

As a new CR method is found using the IBC (Call CR Method in other class) and IBC (Method Called by CR Method in other class) in a NSC, a NCR method in the class may come to be a CR method. A malicious code in a NSC can change the value of a variable referenced by a CR method in the class. A method M can be a CR method if the IBC holds:

IBC (Change to Variable Referenced by CR Method in NSC): *A method M is defined in a NSC and it changes the value of a variable referenced by a CR method in the class.*

4.3.4. Call CR method in NSC

A malicious method in a NSC can call a CR method in the class. A method M in the following IBC can be a CR method if the IBC holds:

IBC (Call CR method in NSC): *A method M is defined in a NSC and it calls a CR method in the class.*

4.3.5. Method Called by a CR method in NSC

A method in a NSC, which is called by a CR method in the class, can contain a malicious code. A method M in the following IBC can be a CR method if the IBC holds:

IBC (Method Called by a CR method in NSC): *A method M is defined in a NSC and it is called by a CR method in the class.*

4.3.6. Inner Class of NSC

For the inner classes of a NSC, NSC needs to apply the following IBCs for NSC similar to those for Secure Class:

- ***IBC (Call CR Method in Inner Class of NSC)***
- ***IBC (Change to Variable Referenced by Inner Class of NSC)***
- ***IBC (Method Called by Inner Class of NSC)***

4.3.7. Additional IBCs for SC, SDWC, and NSDWC

As data or stamp coupling finds additional CR methods in SC, SDWC, and NSDWC, those classes may need to be analyzed again by means of the IBCs defined in sections 5.1 and 5.2.

4.4. Code Review Scope with Subclass Coupling in a Single Package

Subclass coupling is an inheritance relationship between a parent class and its child class in object-oriented programming in which a child class inherits its parent class. The inheritance relationship is a one-way coupling where a child class can access the members (e.g., variables and methods) of a parent class, but the reverse is not true. A child class reads and updates the value of a variable declared with the protected access modifier in its parent class, and it calls a method declared with the protected access modifier in its parent class. Also, there is a polymorphism between a child class and its parent class in which a variable or method declared in the parent class can be overridden by the same variable or method name in the child class.

An assumption for subclass coupling is that there will not be any inner classes declared by classes that are engaged in this type of coupling.

Suppose that the IBCs specified for content, external, and data/stamp (CED) couplings in sections 5.1 through 5.3 are first applied to all classes in a program including parent and child classes, and then the following IBCs for subclass coupling are applied to the program. Fig. 4 depicts the child class and parent class in subclass coupling in which the parent class contains the private pSecVar1 and protected pSecVar2 secure variables, the private pNonsecVar1 and protected pNonsecVar2 non-secure variables, and the private pOperation1(), public pOperation2(), and protected pOperation3() methods.

4.4.1. Change Protected Secure Variable in Parent

A child class can access a protected variable in its parent class, which may be a secure data that requires the integrity security. A malicious code can be hidden in a method in a child class, which tampers the value of a protected secure variable in a parent class. A method M in a child class in the following IBC can be a CR method if the IBC holds:

IBC (Change Protected Secure Variable in Parent): *A method M in a class changes the value of a protected secure variable (SV) in its parent class (e.g., pSecVar2 in Fig. 4).*

4.4.2. Change Protected Non-Secure Variable in Parent

A parent class allows its child class to access the protected non-secure variables defined in a parent class. If the non-secure variable is referenced by a CR method in the parent class, the method in the child class may hide a malicious code. A method M in a child class in the following IBC can be a CR method if the IBC holds:

IBC (Change Protected Non-Secure Variable in Parent): *A method M in a class changes the value of a protected NSV in its parent (e.g., pNonsecVar2 in Fig. 4) and any CR method in the parent class uses the protected NSV.*

The IBCs for data/stamp coupling can find a NCR method in a child class that calls a protected CR method (e.g., pOperation3() in Fig. 4) in the parent using the class inheritance. In the case, the NCR method can be a CR method, and it is not necessary to specify a new IBC for this case in subclass coupling.

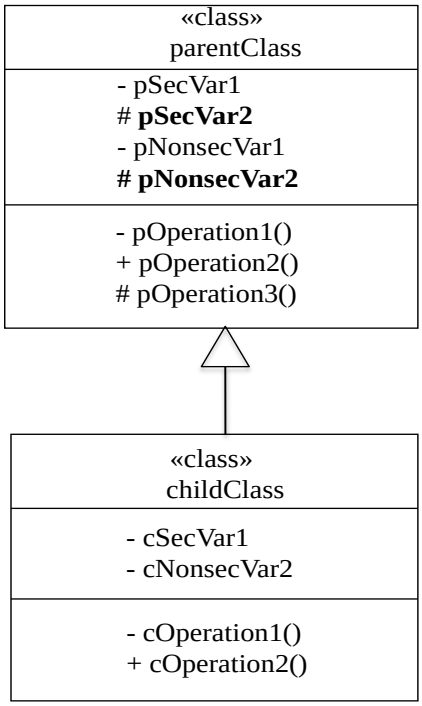


Figure 4 Child and Parent Classes in Subclass Coupling

4.4.3. Access Protected Member (Variable and Method) from Other Classes

In general, a protected member (e.g., variable or method) in a parent class is designed to allow its child class to access the member using the inheritance mechanism. Even though

a class is not a child class of a parent class, syntactically it may access the protected member declared in the parent class. However, it is not the intention for designing a protected member in a parent class. A malicious code can be added to a class that is not a child class of a parent class, so that the code accesses a protected member in a parent class. The malicious code might tamper the value of a protected variable or affect the value of a secure variable in the parent class. In this dissertation, we assume that a class, which is not a child class of a parent class, is not allowed to access a protected member declared in a parent class. A method M in a class in the following IBC can be a CR method if the IBC holds:

IBC (Access Protected Member from Other Classes): *A method M is defined in a class, and the class is not a child class, and the method M accesses a protected member (e.g., protected variable or method) in a parent class.*

This IBC includes a method M in a child class, which accesses a protected variable or method that is not defined in its parent class.

4.5. Code Review Scope with Package Coupling

The IBCs specified in sections 5.1 through 5.4 aim at finding the scope of code review for a single package, which can be extended to multiple packages. There are additional IBCs for multiple packages as a package imports other packages. When a package imports another package, the methods in the importing package can call the methods defined in a class of the imported package. Reversely, if both packages import each other, a method in the imported packages can invoke the methods in the importing package. For class inheritance, a child class in the importing package can access the protected

members defined in its parent class in the imported package. The reverse is also true if there is two-way import of packages.

4.5.1. Call CR Method in another Package

When a package imports another package, a method in a class in the importing package can call a method in a class in an imported package. Also, the reverse is true if both packages import each other. A malicious code can be hidden in a method in a class in a package, which calls a CR method in a class in another package, so that the integrity security is breached. A method M in the following IBC can be a CR method if the IBC holds:

IBC (Call CR Method in another Package): *There is a package import relationship between two packages, and a method M is defined in a class in a package, and it calls a CR method in a class in another package.*

4.5.2. Called by CR Method in another Package

When there is a package import relationship between two packages, a CR method in a package can call a NCR method in another package. A malicious code can be hidden in a NCR method in a class in a package, which is called by a CR method in a class in another package, so that the integrity security is breached. A method M in the following can be a CR method if the IBC holds:

IBC (Called by CR Method in another Package): There is a package import relationship between two packages, and a method M is defined in a class in a package, and it is called by a CR method in a class in another package.

4.6. Dynamic Code Review Scope

The dynamic analysis addresses the runtime execution starting from the main() method in a Java object oriented program. It is the part of the program that is executed, creates objects and operations on them at run time. Make no mistake this analysis is still before compile time.

We will enumerate the conditions that have been identified and present them in OCL in a later chapter. Additionally, the data that is needed for this analysis is specified along with algorithms to create the data and subsequently process the data. One difference between all other analysis discussed earlier and the one for dynamic or main() method is that in this analysis, we will not be identifying any new CR methods, rather we will point the constructs in the main or other methods that create objects that are secure or objects that call CR methods during execution.

4.6.1. Creation of a secure object

If an object is created in the main() or any other method and the object is of a secure class, then the object creation is a security spot.

IBC (Secure Object Created): *A new object of a secure class is created in a method at runtime.*

4.6.2. Creation of a secure object using a secure constructor

If an object is created in the main() or another method and the object is of a secure class, and the constructor of the class is a CR method, then the object creation/constructor is a security spot.

IBC (Secure Object created using secure constructor): *A new object of a secure class is created using the class constructor which is a CR method.*

4.6.3. An object calls a secure method

If an object is created in the main() or any other method and the object is of a secure class or non-secure class, and if the object calls a method which is a CR method, then the object method call is a security spot.

IBC (Object calls a CR method): *An object calls a method which is a CR method.*

4.6.4. Copy of a secure object to another object (Aliasing)

If an object is created in the main() or any other method and the object is of a secure class, then, if that object is copied into another object (aliased) then the object copy is a security spot.

IBC (Object copied into another object): *An object of secure class is copied into another object*

CHAPTER 5

5. JAVA META-MODEL AND IBC DEFINITION USING OBJECT CONSTRAINT LANGUAGE

A Meta-model for a programming language is a model that depicts the syntax for the language. For the purpose of this dissertation, we will limit ourselves to the object-oriented programming language Java. We will work with the core java elements. The meta-model for a target program language, such as the Java, aids in defining the integrity breach conditions by means of the object constraint language (OCL) [Warmer03]. The meta-model for a target program language describes the semantic relationships between the program elements (e.g., class or variable) of the program language. The integrity breach conditions for a target program language can be formally specified by means of the object constraint language (OCL) with the meta-model. The OCL is a standardized notation for specifying the constraints on the elements in a model. The integrity breach conditions are defined by taking into consideration the relationships between meta-classes in the meta-model.

5.1. Java Meta-Model

The Java language is selected as a target program language in this dissertation to implement and validate our approach. The program elements (such as variable and class) and their relationships in a Java program are instances of meta-classes and their relationships in the meta-model for the Java language. The integrity breach conditions described using the OCL at the Java meta-model level are applied to each Java program, which is an instance of the meta-model. In this dissertation, we assume that a target Java program is a sequential program. There are different types of programs such as

sequential, concurrent or parallel programs, but this dissertation focuses on the sequential programs to simplify our approach. Our approach could be extended to other programming types.

Fig. 5 depicts a meta-model for the Java programming language. A Program meta-class contains one or more Package meta-classes, each of which can import zero or more other Package meta-classes. A Package meta-class can contain one or more Class meta-classes, and zero or more Interface meta-classes. A Class meta-class can contain zero or more other Class meta-classes as its inner classes. A Class meta-class can extend another Class meta-class. A Class meta-class can contain zero or more Variable meta-classes and zero or more Method meta-classes.

A Class meta-class instantiates one or more Object meta-classes, each of which can call another Object meta-class. An Interface meta-class extends zero or more other Interface meta-classes. A Class meta-class implements its Interface meta-class. An Interface meta-class can contain zero or more Method meta-class. A Class meta-class can modify or reference a DB Table meta-class.

The DB Table meta-class does not belong to Java, but it is included in the Java meta-model for describing our approach.

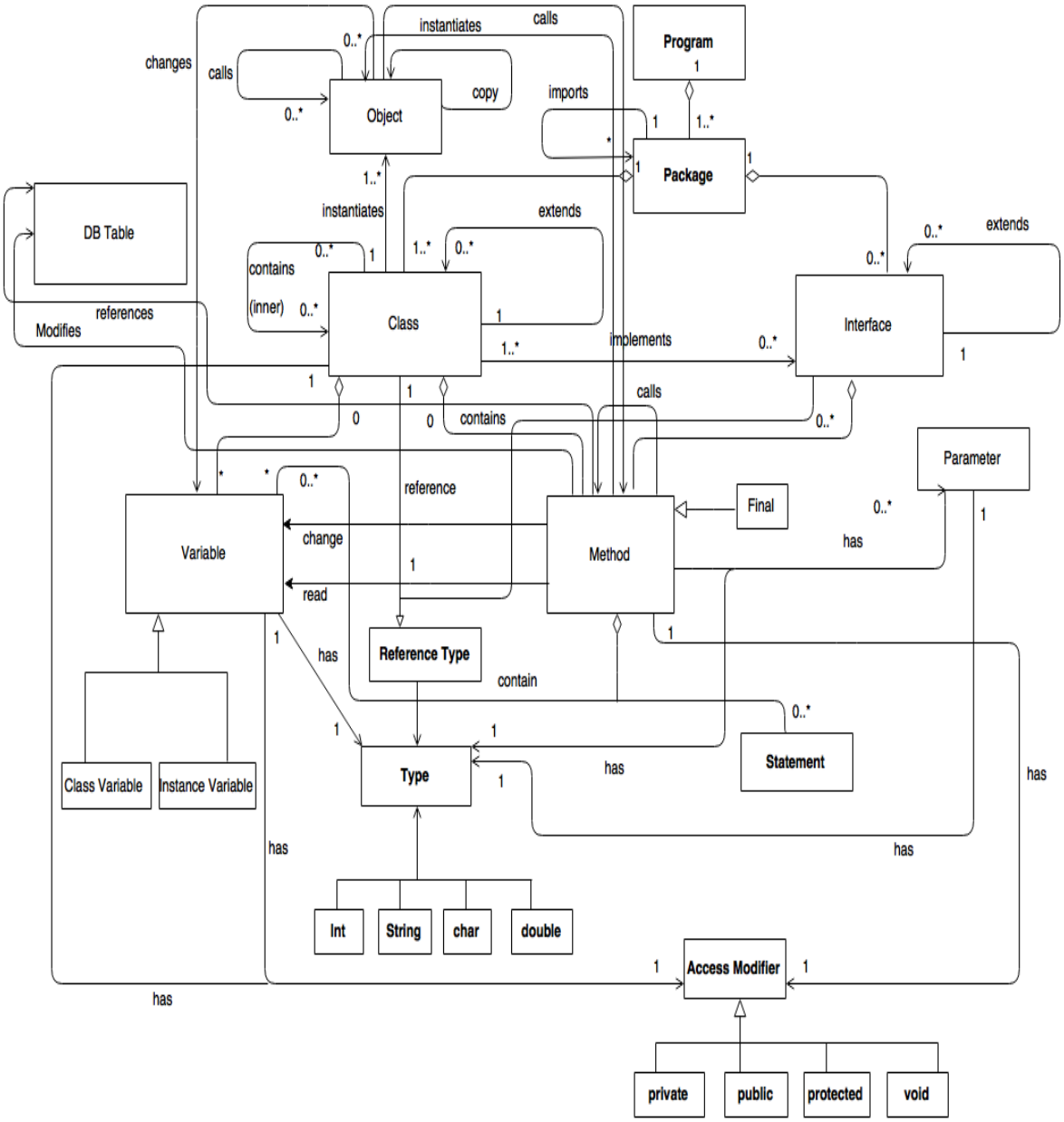


Figure 5 A Java Meta-Model

A Method meta-class may have multiple Variable meta-classes, and it may also have multiple Statement meta-classes. A Method meta-class may have multiple Parameter meta-classes, and a Parameter meta-class has a Type meta-class. A method meta-class can change or read a variable meta-class.

A Method meta-class has a Type meta-class, which can be a specialized to Integer, Double, Char, or String meta-classes. A Variable meta-class can be specialized to a Class variable or Instance variable meta-class. A Class, Method, or Variable meta-class has an Access Modifier meta-class, which can be specialized to a Private, Public, Protected, or Void meta-class.

5.2. Java Meta-Classes

Fig. 6 depicts the attributes of the Java meta-classes in the Java meta-model depicted in Fig. 5.

The Program meta-class has a name represented by a String.

The Package meta-class has a name represented by a String.

The Class meta-class has a name that is represented by a String, a Stereotype1 setting with 2 settings, Secure or Non-Secure, a Type setting with 3 settings, Parent/Child/Inner, and a Stereotype2 setting with 2 settings, DWC or NDWC.

The Object meta-class has a name represented by a String, a Stereotype1 with 2 settings, Secure or Non-Secure, and a Stereotype2 setting with 2 settings, CR or NCR.

The Variable meta-class has a name represented by a String. It also has a Stereotype1 of secure or non-secure depending upon whether the Variable is a secure variable or non-secure variable, a Stereotype2 setting for Class variable or Instance variable, and a Value that can be set to a value in a range.

The Method meta-class has a name represented by a String and a Stereotype of Code Review or Non Code Review.

The Interface meta-class has a name represented by a String.

The Type meta-class has a type set to reference and a Stereotype of Integer, double, char or string.

The DB Table meta-class has a name represented by a string and a type that can be set to Secure or Non-Secure.

The Access Modifier meta-class has a Stereotype that can be set to Private, Public, Protected or Void.

The Statement meta-class has a LineNum that can be set to a value in the range 1 through total lines of source code.

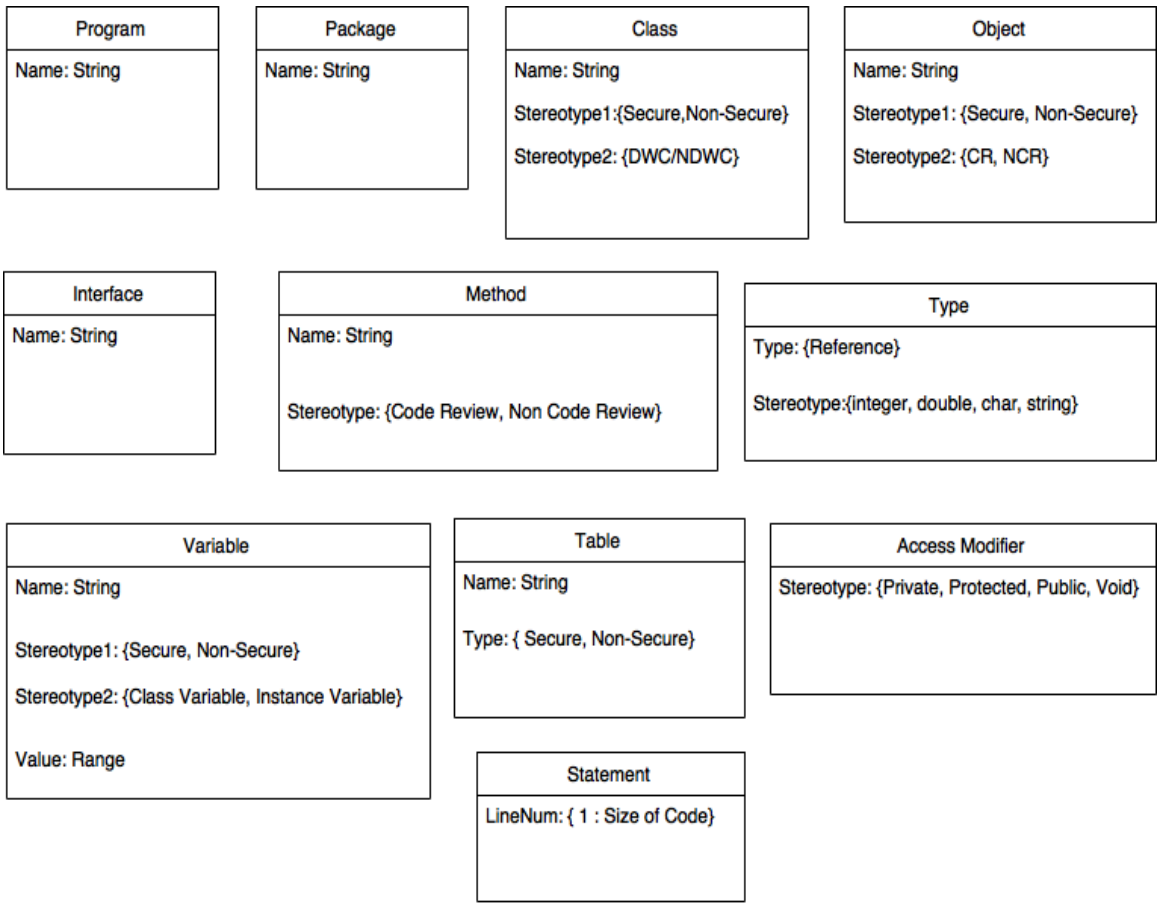


Figure 6 Attributes of Java Meta-Classes

5.3. Conditions described in Object Constraint Language

This section will detail all the IB conditions and express them in the OCL. These expressions in OCL are then converted to implementation in the Tool design in section 9.

We have an assumption about the class variables. They must be declared as private or protected. This assumption is checked using an Assumption condition mentioned below

Assumption Condition (Class Variable declaration): *A variable V is defined in a class, and it is defined as private or protected.*

Condition expressed in OCL:

```
Context Class inv:
(Self.variable.AccessModifier.Stereotype = 'private' or
'protected')
```

5.4. Conditions in OCL for Code Review Scope with Content Coupling

5.4.1. Direct Change in Secure Class

IBC (Direct Change in Secure Class): *A method M is defined in a secure class, and it changes the value of a secure variable in the class.*

```
Context Class inv:

Self.Method->forAll (m Method | (self.Stereotype1 = 'Secure') and
(m.changes.Stereotype2 = 'Class Variable' or 'Instance Variable')
and (m.changes.Stereotype1 = 'Secure')) implies (m.Stereotype =
'Code Review')
```

5.4.2. Change to Variable Referenced by CR Method in Secure Class

IBC (Change to Variable Referenced by CR Method in Secure Class): *A method M is defined in a secure class, and it changes the value of a non-secure variable in the class that is referenced by a CR method in the class to modify the value of a secure variable.*

```
Context Class inv:
Self.Method -> forAll (m1, m2: Method| (Self.Stereotype1 =
'Secure') and (Self.Variable.Stereotype1 = 'Non-Secure') and
(Self.Variable.Stereotype2 = 'Class Variable' or 'Instance
```

```
Variable') and (m1.changes = Self.Variable) and (m2.Stereotype =
'Code Review') and (m2.reads = Self.Variable)) implies
(m1.Stereotype = 'Code Review')
```

5.4.3. Call CR Method in Secure Class

IBC (Call CR Method in Secure Class): *A method M is defined in a secure class, and it calls a CR method in the class to change the value of a secure variable.*

```
Context Class inv:
Self.Method -> forAll(m: Method | (Self.Stereotype1 = 'Secure')
and (m.Stereotype = 'Non-Code Review') and (m.calls.Stereotype =
'Code Review')) implies (m.Stereotype = 'Code Review')
```

5.4.4. Method Called by CR Method in Secure Class

IBC (Method Called by CR Method in Secure Class): *A method M is defined in a secure class, and it is called by a CR method.*

```
Context Class inv:
Self.Method -> forAll (m1, m2: Method | (Self.Stereotype1 =
'Secure') and (m1.Stereotype = 'Non-Code Review') and
(m2.Stereotype = 'Code Review') and (m2.calls = m1)) implies
(m1.Stereotype = 'Code Review')
```

5.4.5. Call CR Method in Inner Class of Secure Class

IBC (Method in Inner class of Secure Class): *A method M is defined in an inner class of a secure class.*

Associated OCL construct to describe the condition:

```
Context Class inv:
Context Class inv:
Self.Method -> forAll( m: Method | ((Self.Stereotype1 = ‘Secure’)
and (Self.contains.Method.Name = m.Name)) implies (m.Stereotype =
‘Code Review’)
```

IBC (Call CR Method in Inner Class of Secure Class): *A method M is defined in a secure outer class, and it calls a method in an inner class of the secure outer class.*

```
Context Class inv:
((Self.Stereotype1 = ‘Secure’) and (Self.Method.calls =
Self.contains.Method)) implies (Self.Method.Stereotype = ‘Code
Review’)
```

5.4.6. Change to Variable Referred by Inner Class of Secure Class

IBC (Change to Variable Referred by Inner Class of Secure Class): *A method M is defined in a secure outer class, and it changes the value of a variable in the class that is referred by a method in an inner class of the secure outer class.*

```
Context Class inv:
```

```
((Self.Stereotype1 = ‘Secure’) and (Self.Method.changes =
Self.contains.Method.reads)) implies (Self.Method.Stereotype =
‘Code Review’)
```

5.4.7. Method Called by Inner Class of Secure Class

IBC (Method Called by Inner Class of Secure Class): *A method M is defined in a secure outer class, and it is called by a CR method in an inner class of the secure outer class.*

```
Context Class inv:
((Self.Stereotype1 = ‘Secure’) and (Self.Method =
Self.contains.Method.calls)) implies (Self.Method.Stereotype =
‘Code Review’)
```

5.5. Conditions in OCL for Code Review Scope with External Coupling

5.5.1. Direct Change to Data in DWC

IBC (Change to Secure Data in DWC): *A method M is defined in a DWC and it changes a secure data value stored in a database..*

```
Context Class inv:

Self.Method->forAll( m: Method | (self.stereotype2 = ‘DWC’) and

(m.modifies.stereotype = ‘Secure’)) implies (m.Stereotype = ‘Code
Review’)
```

5.5.2. Call CR Method in DWC

IBC (Call CR Method in DWC): *A method is defined in a DWC and it calls a CR method in the class.*

```
Context Class inv:

Self.Method -> forAll(m: Method | (Self.Stereotype1 = 'Secure') and (Self.Stereotype2 = 'DWC') and (m.Stereotype = 'Non-Code Review') and (m.calls.Stereotype = 'Code Review')) implies (m.Stereotype = 'Code Review')
```

5.5.3. Method Called by CR Method in DWC

IBC (Method Called by CR Method in DWC): *A method M is defined in a DWC and it is called by a CR method in the class.*

```
Context Class inv:

Self.Method -> forAll (m1, m2: Method | (Self.Stereotype1 = 'Secure') and (Self.Stereotype2 = 'DWC') and (m1.Stereotype = 'Non-Code Review') and (m2.Stereotype = 'Code Review') and (m2.calls = m1)) implies (m1.Stereotype = 'Code Review')
```

5.6. Conditions in OCL for Code Review Scope with Data/Stamp Coupling

5.6.1. Call CR Method in other class

IBC (Call CR Method in other class): *A method M is defined in a class and it calls a CR method defined in another class.*

```
Context Package inv:

Self.Class -> forAll (c1, c2: Class| (c1.Name != c2.Name) and
(c1.Method.Stereotype = 'Code Review') and (c2.Method.calls =
c1.Method)) implies (m2.Stereotype = 'Code Review')
```

5.6.2. Method Called by CR Method in other class

IBC (Method Called by CR Method in other class): *A method M is defined in a class and it is called by a CR method in another class.*

```
Context Package inv:

Self.Class -> forAll (c1, c2: Class| (c1.Name != c2.Name) and
(c2.Method.Stereotype = 'Code Review') and (c2.Method.calls =
c1.Method)) implies (c1.Method.stereotype = 'Code Review')
```

5.6.3. Change to Variable Referenced by CR Method in NSC

IBC (Change to Variable Referenced by CR Method in NSC): *A method M is defined in a NSC and it changes the value of a variable referenced by a CR method in the class.*

```
Context Class inv:

Self.Method -> forAll (m1, m2: Method| (Self.Stereotype1 = 'Non-
Secure') and (Self.Variable.Stereotype1 = 'Non-Secure') and
(Self.Variable.Stereotype2 = 'Class Variable' or 'Instance
Variable') and (m1.changes = Self.Variable) and (m1.Stereotype =
'Non-Code Review') and (m2.Stereotype = 'Code Review') and
```

```
(m2.reads.Name = Self.Variable.Name)) implies (m1.Stereotype =
‘Code Review’)
```

5.6.4. Call CR method in NSC

IBC (Call CR method in NSC): *A method M is defined in a NSC and it calls a CR method in the class.*

```
Context Class inv:

Self.Method -> forAll (m: Method | (Self.Stereotype1 = ‘Non-
Secure’) and (m.Stereotype = ‘Non-Code Review’) and
(m.calls.Stereotype = ‘Code Review’)) implies (m.Stereotype =
‘Code Review’)
```

5.6.5. Method Called by a CR method in NSC

IBC (Method Called by a CR method in NSC): *A method M is defined in a NSC and it is called by a CR method in the class.*

```
Context Class inv:

Self.Method -> forAll (m1, m2: Method | (Self.Stereotype1 = ‘Non-
Secure’) and (m1.Stereotype = ‘Non-Code Review’) and
(m2.Stereotype = ‘Code Review’) and (m2.calls = m1)) implies
(m1.Stereotype = ‘Code Review’)
```

5.6.6. Call CR Method in Inner Class of Non Secure Class

IBC (Method in Inner class of Secure Class): *A method M is defined in an inner class of a non secure class.*

```
Context Class inv:
Self.Method -> forAll( m: Method | ((Self.Stereotype1 = 'Non-Secure') and (Self.contains.Method = m)) implies (m.Stereotype = 'Code Review'))
```

IBC (Call CR Method in Inner Class of Non Secure Class): *A method M is defined in a non secure outer class, and it calls a method in an inner class of the non secure outer class.*

```
Context Class inv:
((Self.Stereotype1 = 'Non Secure') and (Self.Method.calls = Self.contains.Method)) implies (Self.Method.Stereotype = 'Code Review')
```

5.6.7. Change to Variable Referred by Inner Class of Non Secure Class

IBC (Change to Variable Referred by Inner Class of Non Secure Class): *A method M is defined in a non secure outer class, and it changes the value of a variable in the class that is referred by a method in an inner class of the non secure outer class.*

```
Context Class inv:
```

```
((Self.Stereotype1 = 'Non Secure') and (Self.Method.changes =
Self.contains.Method.reads)) implies (Self.Method.Stereotype =
'Code Review')
```

5.6.8. Method Called by Inner Class of Secure Class

IBC (Method Called by Inner Class of non Secure Class): *A method M is defined in a non secure outer class, and it is called by a CR method in an inner class of the non secure outer class.*

```
Context Class inv:

((Self.Stereotype1 = 'Non Secure') and (Self.Method =
Self.contains.Method.calls)) implies (Self.Method.Stereotype =
'Code Review')
```

5.6.9. Additional IBCs for SC, SDWC, and NSDWC

As data or stamp coupling finds additional CR methods in SC, SDWC, and NSDWC, those classes may need to be analyzed again by means of the IBCs defined in sections 5.1 and 5.2.

5.7. Conditions in OCL for Code Review Scope with Subclass Coupling in a Single Package

5.7.1. Change Protected Secure Variable in Parent

IBC (Change Protected Secure Variable in Parent): *A method M in a class changes the value of a protected secure variable (SV) in its parent class (e.g., pSecVar2 in Fig. 4).*

```
Context Class inv:

Self.Method -> forAll (m method |
(self.extends.Variable.Stereotype1 = ‘Secure’) and
(self.extends.Variable.Stereotype2 = ‘Class Variable’ or
‘Instance Variable’) and
(self.extends.Variable.AccessModifier.Stereotype = ‘Protected’)
and (m.changes = self.extends.Variable)) implies
(m.Stereotype=’Code Review’)
```

5.7.2. Change Protected Non-secure Variable in Parent

IBC (Change Protected Non-secure Variable in Parent): *A method M in a class changes the value of a protected NSV in its parent (e.g., pNonsecVar2 in Fig. 4) and any CR method in the parent class uses the protected NSV.*

```
Context Class inv:

 Self.Method -> forAll (m method |
(self.extends.Variable.Stereotype1 = ‘Non-Secure’) and
(self.extends.Variable.Stereotype2 = ‘Class Variable’ or
‘Instance Variable’) and
(self.extends.Variable.AccessModifier.Stereotype = ‘Protected’)
and (m.changes = self.extends.Variable) and
(self.extends.Method.Stereotype = ‘Code Review’) and
(self.extends.Method.reads = m.changes) implies
(m.Stereotype=’Code Review’)
```

5.7.3. Access Protected Member (Variable and Method) from Other Classes

IBC (Access Protected Member from Other Classes): *A method M is defined in a class, and the class is not a child class, and the method M accesses a protected member (e.g., protected variable or method) in a parent class.*

```
Context Package inv:

Self.Class -> forAll (c1, c2: Class| (c1.Name != c2.Name) and
(c1.extends != c2) and ((c2.Method.AccessModifier = 'Protected')
and (c2.Variable.AccessModifier = 'Protected') and
((c1.Method.calls = c2.Method) or (c1.Method.changes =
c2.Variable))) implies (c1.Method.stereotype = 'Code Review')
```

5.8. Conditions in OCL for Code Review Scope with Package Coupling

5.8.1. Call CR Method in another Package

IBC (Call CR Method in another Package): *There is a package import relationship between two packages, and a method M is defined in a class in a package, and it calls a CR method in a class in another package.*

```
Context Program inv:

Self.Package -> forAll ( p1, p2: Package| (p1 != p2) and
(p1.imports.Name = p2.Name) and (p2.Class.Method.Stereotype =
'Code Review') and (p1.Class.Method.Stereotype = 'Non-Code
```

```
Review’) and (p1.Class.Method.calls = p2.Class.Method) implies
(p1.Class.Method.Stereotype = ‘Code Review’)
```

5.8.2. Called by CR Method in another Package

IBC (Called by CR Method in another Package): *There is a package import relationship between two packages, and a method M is defined in a class in a package, and it is called by a CR method in a class in another package.*

```
Context Program inv:
Self.Package -> forAll ( p1, p2: Package| (p1 != p2) and
(p1.imports.Name = p2.Name) and (p2.Class.Method.Stereotype =
‘Code Review’) and (p1.Class.Method.Stereotype = ‘Non-Code
Review’) and (p2.Class.Method.calls = p1.Class.Method) implies
(p1.Class.Method.Stereotype = ‘Code Review’)
```

5.9. Conditions in OCL for Code Review Scope with main()

5.9.1. Creation of a secure object in main()

IBC (Secure Object Created): *A new object of a secure class is created in the main() routine.*

```
Context Class inv:
(Self.Stereotype1 = ‘Secure’) implies
(Self.Method.Object.Stereotype2 = ‘Code Review’)
```

5.9.2. Creation of a secure object in main() using a secure constructor

IBC (Secure Object created using secure constructor): *A new object of a secure class is created using the class constructor which is a CR method.*

```
Context Class inv:
(Self.Stereotype1 = 'Secure') implies
(Self.Method.Object.Stereotype2 = 'Code Review')
```

5.9.3. An object in main() calls a secure method

IBC (Object calls a CR method): *An object calls a method which is a CR method.*

```
Context Object inv:
(Self.Calls.Stereotype = 'Code Review') implies (Self.Stereotype
= 'Code Review')
```

5.9.4. Copy of a secure object in main() to another object (Aliasing)

IBC (Object copied into another object): *An object of secure class is copied into another object*

```
Context Class inv:
(Self.Stereotype1 = 'Secure') implies
(Self.Object.copy.Stereotype = 'Code Review')
```

CHAPTER 6

6. TOOL SUPPORT

A tool is developed for supporting the CAIS approach proposed in this dissertation, and used for case studies to validate the approach. The tool for the CAIS is fully automated to analyze Java codes by means of the integrity breach conditions so that it identifies the possible methods of classes that might be compromised by an insider attack. The tool is implemented with the Python programming language.

The changeability or evolution is a primary concern for designing the tool supporting the CAIS. Although we have exhaustively enumerated the integrity breach conditions that are criteria for identifying possible malicious codes in a program, new integrity breach conditions might be found later. Also, additional integrity breach conditions might need to be added to the tool as the tool evolves for programs specific to applications or for reflecting new technologies.

6.1. Structure of the Tool

The tool for CAIS is designed to support the changeability or evolution. Fig. 7 depicts the overview of tool for the CAIS that is structured with the following components:

1) GUI: The GUI is used to input the list of secure data (variables or tables) encapsulated in secure classes or SDWC of a program. Each secure data in a secure class is specified with the location information such as package, class, and variable in a program whereas a secure data in a SDWC is described as a secure database table that contains secure data along with the SDW class which accesses the table.

2) Code scanner. The code scanner extracts the integrity security relations from an application program. The starting point of the extraction is by referring to the secure data entered by a user through the GUI. Each integrity security relation contains the elements of a program such as class name, method and its arguments, or variables. The integrity security relations are pre-determined by analyzing each IBC as to what information needs to be extracted from a program for checking the condition. Rather than checking the IBCs using the program directly, each IBC is checked using the integrity security relations extracted by the scanner from a program. Thus, the integrity security relations and the program are one-to-one relationship from the integrity security analyzer perspective. The integrity security relations can be also reused for checking a new IBC that might be added to the tool later. When the new IBC requires additional relations that are not in the existing integrity security relations, the scanner is changed only to have a module for extracting the relations from a program.
3) Integrity security relation database. The integrity security relation database is a repository for storing the integrity security relations and the secure data in a program. The database is a relational database containing relations in a table format. The SQLite3 database is identified and used for this dissertation. The integrity security relation database is a collection of 20 tables each of which are described separately in this dissertation.
4) Integrity security analyzer. The integrity security analyzer evaluates each IBC and analyzes the data in the integrity security relation database using the breach conditions. It applies each IBC sequentially in a certain order to identify all

possible security spots that might hide malicious code in a program. The analyzer may execute the sequence more than once if it finds new CR methods in a sequence pass. It generates the scope of code review for the integrity security of a program, which contains the list of CR methods in the program. The components that analyze each IBC are designed independently of each other. Although a new IBC is added to or an IBC is changed to the tool, this change does not give any impact on the components checking other IBCs.

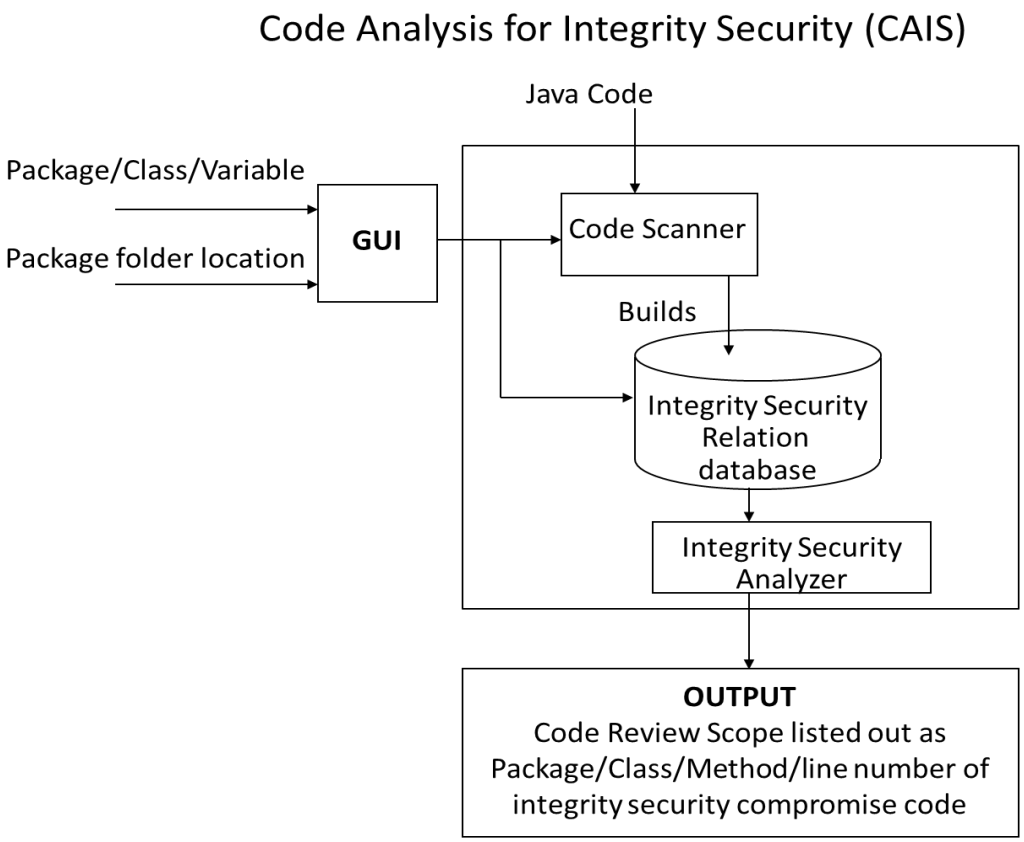


Figure 7 Schematic of the CAIS Tool

6.2. The Code Scanner

The code scanner is described in this section along with the scanning algorithm that enables the scanner to complete a scan of the java source code.

The scanner is provided the location of the packages of the application. It is assumed that all packages are in a single location/folder. The code scanner is responsible for scanning all of the source code files from each of the packages of the application. The scan is a single scan of each of the source code files. The scanner builds Security Integrity Relations Database by populating one or more of the 20 tables that comprise the database. To build the tables the scanner matches the tokens scanned to the specific constructs required to determine which table to populate. The algorithms for the scanner are described in the section 6.2.2 and 6.3

Appendix section A (iii) provides a schematic of the CodeScanner module that depicts the other functions called and the logic of those functions. The section on the Integrity Security Relations Database (6.3) describes each unique table.

6.2.1. List of distinct tables built by the CodeScanner

The tables (17 in all) built by the CodeScanner are as below:

1) T1_SecureVar - Contains “pkg, Class, SecureVariable/Table”
2) T2_PkgCls - Contains “pkg, Class, LineNum”
3) T3_Var_Modifier - Contains “pkg, Class, Var, Modifier, LineNum”
4) T4_ChangedVar - Contains “pkg, Class, Method, ChangedVar, LineNum”
5) T5_UsedVar - Contains “pkg, Class, Method, UsedVar, LineNum”

6) T6_MethodCall - Contains "pkg, Class, Method, pkg2, Class2, CalledMethod,

LineNum"

7) T7_InnClsChgdVar - Contains "pkg, Class, InnerClass, ICM, ChangedVar, LineNum"
8) T8_InnClsUsedVar - Contains "pkg, Class, InnerClass, ICM, UsedVar, LineNum"
9) PKGCLSMTD - Contains "pkg, Class, Method, Qualifier, Return"
10) PKGIMPORT - Contains "pkg, importedpkg, javafile, LineNum"
11) Extends - Contains "pkg, ParentClass, InheritedClass"
12) InnerClass - Contains "pkg, ParentClass, InnerClass, LineNum"
13) SecureClass - Contains "pkg, Class"
14) ObjectMethod - Contains "pkg, Class, Object, Method, LineNum"
15) ObjectMethodCall - Contains "pkg, Class, Object, Method, LineNum"
16) ObjectCopy - Contains "pkg, Class, Object1, Object2, LineNum"
17) MainClass - Contains "pkg, Class"

6.2.2. Code Scanning Algorithm

Usage: CodeScanner(secfile, pkgloc, pkglist)

Inputs: secfile contains the text file that contains the secure variables list

pkgloc contains the full path of the packages location

pkglist contains the list of packages for the application

Output: None

i. CodeScanner sets all the global variables needed for the scanning process

ii. CodeScanner sets up local list data structures to store certain cached database table contents

iii. CodeScanner sets up lists of ignorewords and DBKW (Database Keywords) words which are keywords which will be skipped as needed for certain code scanning. These are either reserved words or system words that are not user program created. The list of keywords is mentioned at the end of this algorithm.

iv. CodeScanner creates all of the database tables required to scan and analyze the code by calling createtables()

v. CodeScanner creates the secure variables list by calling createsecvarlist() and storing the secfile it received in Table T2_PkgCls using BuildTable2() function

vi. CodeScanner outputs basic scanning messages to the Outfile

vii. CodeScanner sets up the list of packages to scan based on pkgloc and pkglist

viii. CodeScanner extracts the names of the files in the packages in the given folder/directory

ix. **FOR Each** file in the package:

- CodeScanner starts the scan of the java source files from the first package. It will scan and process each line of the java source file, strip the line of blanks and splits the tokens into a list of tokens. After each line is scanned it will process the line based on the login given in the bulleted points and continue scanning till it reaches the end of file at which point it will start the scan of the next java source file in that package.

- CodeScanner looks for certain tokens and calls routines to process further based on the tokens read.

- If CodeScanner scans token “package” it will record the package being scanned
- If CodeScanner scans token “import” it will build pkgimport table accordingly
- If CodeScanner scans token “import” and “java” and “sql” it will set DBWC
- If CodeScanner scans tokens “public” and “class” and “classname” for the given source file, then it will Build Table T2_PkgCls
    - If DBWC is True then call HandleDBWClass() for further scanning
    - Else call HandleClass() for further scanning
- If CodeScanner scans token “extends” then it will add parent and child class entries in the Extends Table and call HandleSubClass() to handle further scanning
- CodeScanner scans the next line of source code, strips the line of blanks
- **END FOR**

x. CodeScanner builds the local lists for Table T1_SecureVar by calling BuildTableT1_SecureVarList()

xi. CodeScanner builds the local lists for Table T3_Var_Modifier by calling BuildTableT3_Var_ModifierList()

xii. CodeScanner builds the local lists for Table SecureClass by calling BuildTableSecureClassesList()

xiii. CodeScanner builds the local lists for Table PkgClsMtdList by calling BuildTablePkgClsMtdList()

xiv. CodeScanner writes debug messages and returns()

6.2.3. List of Keywords to be ignored or used

1) ignorewords = ['DB_URL','System','return','Statement','ResultSet','Connection', 'connection','new','PreparedStatement', 'close', 'println', 'sqlstatement', 'sqlStatement', 'SELECT', 'UPDATE', 'INSERT', 'DELETE', 'REPLACE', 'sqlException', 'prepareStatement', 'getColumnCount', 'getMetaData', 'for','while','if','setString', 'setInt', 'nextInt', 'execute', 'executeUpdate', 'executeDelete', 'executeSelect', 'executeReplace']
2) DBKW= ['UPDATE','INSERT','DELETE','REPLACE']

6.3. Integrity Security Relations Database

The Integrity Security Relation Database is built by the code scanner. This sections details the format of each of the table listed in the previous section and describes the algorithm used to create the table.

6.3.1. Scanning Algorithm to Create Table T1_SecureVar

The first step for the CAIS tool is for the Software Architect or the Software Engineer to provide the list of secure variables being used in a package. This information can be supplied as package name, class, secure variable name through the input file. The input file “SV.txt” is read by the Code Scanner and stored in an internal data structure/table T1_SecureVar as below. The architect will also provide the names of the database tables which contain secure data in the java application. Note that there could be more than one secure variable and it could be in different class or package.

The **table columns** are Package to store package of the secure variable, Class to store the class containing the secure variable and SecVar/SecTable to store the secure variable or Secure Table name.

This table is needed to refer to the list of secure variables when the analyzer is processing the tables to identify the methods which change the values in the secure variable or modify the secure table.

Package	Class	SecVar/SecTable
Pkg1	C1	SV1
Pkg2	C2	SV2
..	..	..

Table 1 Table T1_SecureVar

```
// Process when scanning source files for creating Table T1_SecureVar;

Read the secure variables list file provided by Architect;
While not end of input Do
     Read a line of the input;
     Separate the fields and store into a table T1_SecureVar;
End While
```

6.3.2. Scanning Algorithm to Create Table T2_PkgCls

Another table created by the Code Scanner is Table T2_PkgCls which is a table of pkg, class and the line number at which the class is declared.

The **table columns** are Package to store package, Class to store the class and LineNum to determine the Line number when this declaration is encountered in the source file.

This table is built for future use if this information is required. At this time there is no specific requirement.

Package	Class	LineNum
Pkg1	C1	#
Pkg1	C2	#
..	..	..

Table 2 Table T2_PkgCls

```
// Process when scanning the line of java source code
// Add Logic to determine if a class is being declared
Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
   If Class is being declared Then
      Pkg = current pkg code being scanned;
      Class = current class code being scanned;
      LineNum = Line number of the current ScanLine;
      Store 3 entities in table T2_PkgCls (list of 3 fields);
      Scan the next line of source code into ScanLine;
   End If
   Continuation of Logic …
End While
```

6.3.3. Scanning Algorithm to Create Table T3_Var_Modifier

The **table columns** are Package to store package of currently scanned package, Class to store the class, Var to store the Variable being declared, Modifier to record the Modifier associated with the Variable and LineNum to record the line number on which this variable declaration is coded.

This table is needed by the analyzer when processing the tables to identify that the variables in classes are declared as private or protected. We assume that this is required in our assumptions, but there is a check to ensure that a mis-declaration is not overlooked.

Package	Class	Var	Modifier	LineNum
Pkg1	C1	V1	Private	#

Table 3 Table T3_Var_Modifier

```
// Process when scanning the line of java source code
// Add logic to determine if a class variable is being declared
Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
  If a class variable is being declared Then
     Pkg = current pkg code being scanned;
     Class = current class code being scanned;
     Var = variable being declared in ScanLine;
     Mod = access modifier of the variable being declared;
     LineNum = Line number of the current ScanLine;
     Store 5 entities in table T3_Var_Modifier (list of 5
fields);
     Scan the next line of source code into ScanLine;
   End If
   Continuation Logic….
End While
```

6.3.4. Scanning Algorithm to Create Table T4_ChangedVar

The **table columns** are Package to store package of the code being currently scanned, Class to store the class being scanned, Method to store the method being scanned, ChangedVar to store the name of the variable which is changed in the method and LineNum to record the line number of the code occurrence in the source file.

This table is needed by the analyzer when it is checking to see if a variable was changed and if it was of the type secure variable. It is also needed by the analyzer to check if there is indirect modification of a secure variable.

Package	Class	Method	ChangedVar	LineNum
Pkg1	C1	M1	V1	#

Table 4 Table T4_ChangedVar

There are 2 parts to Creating/adding entries to the Table T4_ChangedVar.

1) An assignment statement in the Code

// Process when scanning the line of java source code

```
Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
   If “Var = ” is a construct in the ScanLine Then
      Pkg = current pkg code being scanned;
      Class = current class code being scanned;
      Method = current method code being scanned;
      Var = variable being changed in ScanLine;
      LineNum = LineNumber of the current ScanLine;
      Store the 5 entities in table T4_ChangedVar;
   End If
   Continuation of Logic …
   Scan the next line of source code into ScanLine;
End
```

2) An Update/Modify/Insert/Delete on Secure table statement

// Process when scanning the line of java source code

```
Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
   If “UPDATE/MODIFY/INSERT/DELETE” AND Secure TABLE is a
    construct in the ScanLine Then
      Pkg = current pkg code being scanned;
```

```
      Class = current class code being scanned;
      Method = current method code being scanned;
      TABLE = TABLE being changed in ScanLine;
      LineNum = LineNumber of the current ScanLine;
      Store the 5 entities in table T4_ChangedVar;
   End If
   Continuation of Logic …
   Scan the next line of source code into ScanLine;
End
```

6.3.5. Scanning Algorithm to Create Table T5_UsedVar

The scanner will create table T5_UsedVar that is required by the analyzer by capturing the information as it scans.

The **table columns** are Package to store package being currently scanned, Class to store the class, Method to store the name of method being scanned, UsedVar to store the name of the variable which is referenced in the method, and LineNum to record the line number at which the code is encountered. This table is needed by the analyzer when it is checking for indirect change to a secure variable.

Package	Class	Method	UsedVar	LineNum
Pkg1	C1	M2	V1	#

Table 5 Table T5_UsedVar

```
// Process when scanning the line of java source code

Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
   If "= var" is the construct in ScanLine Then
      Pkg = current pkg code being scanned;
      Class = current class code being scanned;
      Method current method code being scanned;
      Var = variable being used in ScanLine;
      LineNum = LineNumber of the current ScanLine;
```

```
      Store Pkg,Class,Method,Var,LineNum in T5_UsedVar (list of 5
      fields);
   End If
   Continuation of logic …
      Scan the next line of source code into ScanLine;
End
```

6.3.6. Scanning Algorithm to Create Table T6_MethodCall

The **table columns** are Package to store name of package being scanned, Class to store the class, Method to stored method being scanned, Package to store package of the called method, Class to store class of called method, CalledMethod to store the method being called, and LineNum to record the line number in the source file where the method call is encountered.

This table is needed by the analyzer when it determines what CR methods are being called and if any CR method is calling other methods.

Package	Class	Method	Package	Class	CalledMethod	LineNum
Pkg1	C1	M1	Pkg1	C1	M2	#

Table 6 Table T6_MethodCall

```
// Process when scanning the line of java source code

Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
   If "Object.method()" call construct is in ScanLine Then
      Pkg = current pkg code being scanned;
      Class = current class code being scanned;
      Method1 = current method code being scanned;
      LineNum = LineNumber of the current ScanLine;
      Method2 = Method being called;
```

```
      Store     Pkg,Class,Method1,Pkg,Class,Method2,LineNum     in
      T6_MethodCall;
   End If
   Continuation of logic …
   Scan the next line of source code into ScanLine;
End
```

Additional algorithm to update Table T6_MethodCall when pkg of called method is not known during initial scan. Please refer to PKGCLSMTD table information. It is possible that a method being scanned calls a method in a package/class that has not yet been scanned. In this case there is a need to refer to the PKGCLSMTD table after the entire scan is completed. The Package and Class of the called method are updated to reflect the information in the PKGCLSMTD.

```
      // Process when scanning the line of java source code

Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
   If "Object.method()" call construct is in ScanLine Then
      Pkg1 = current pkg code being scanned;
      Class1 = current class code being scanned;
      Method1 = current method code being scanned;
      LineNum = LineNumber of the current ScanLine;
      Class2 = Class of method being called;
      Method2 = Method being called;
      Pkg2 = package of method being called
      Separate algorithm to identify Class2 & Pkg2: Class2 and
      Pkg2 may not be scanned yet as is most likely the case.
      Refer to PkgClsMtd table after the CodeScanner process is
      complete to identify which Pkg and Class this method belongs
      to and update the Table 6 entry for Class2 and Pkg2
      Store    Pkg1,Class1,Method1,Pkg2,Class,Method2,LineNum    in
T6_MethodCall;
   End If
   Continuation of logic …
   Scan the next line of source code into ScanLine;
End While
```

6.3.7. Scanning Algorithm to Create Table T7_InnClsChgdVar

The **table columns** are Package to store package being scanned, Class to store the class , InnerClass to store the Inner Class being scanned, InnerClassMethod to store the InnerClassMethod being scanned, ChangedVar to store the name of the variable that is changed in the line of code and LineNum to store the line number where this construct is encountered in the source file.

This table is needed by the analyzer when it is checking to see if a variable was changed and if it was of the type secure variable. It is also needed by the analyzer to check if there is indirect modification of a secure variable.

Package	Class	InnerClass	InnerClassMethod	ChangedVar	LineNum
Pkg1	C1	IC1	IC1M1	V1	#

Table 7 Table T7_InnClsChgdVar

```
// Process when scanning the line of java source code

Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
   If "Var =" construct found when InnerClass method is being
   processed Then
      LineNum = LineNumber of the current ScanLine;
      Pkg = current pkg code being scanned;
      Class = current class code being scanned;
      InnerClass = innerclass code being scanned;
      ICMethod = innerclass method code being scanned;
      ChangedVar = variable being changed in ScanLine;
      Store  Pkg,Class,InnerClass,ICMethod,ChangedVar,LineNum  in
      T7_InnClsChgdVar;
   End If
   Continuation of logic …
```

```
   Scan the next line of source code into ScanLine;
End While
```

6.3.8. Scanning Algorithm to Create Table T8_InnClsUsedVar

The **table columns** are Package to store package being scanned, Class to store the class, InnerClass to store the Inner Class being scanned, InnerClassMethod to store the InnerClassMethod being scanned, UsedVar to store the name of the variable that is referenced in the line of code and LineNum to store the line number where this construct is encountered in the source file.

This table is needed by the analyzer when it is checking to see if there is indirect modification of a secure variable.

Package	Class	InnerClass	InnerClassMethod	UsedVar	LineNum
Pkg1	C1	IC2	IC2M2	V1	#

Table 8 Table T8_InnClsUsedVar

```
// Process when scanning the line of java source code

Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
   If "= Var" construct found when InnerClass method is being
   processed Then
      LineNum = LineNumber of the current ScanLine;
      Pkg = current pkg code being scanned;
      Class = current class code being scanned;
      InnerClass = innerclass code being scanned;
      ICMethod = innerclass method code being scanned;
      ReadVar = variable being changed in ScanLine;
      Store    Pkg,Class,InnerClass,ICMethod,ReadVar,LineNum    in
      T8_InnClsUsedVar;
   End If
   Continuation of logic …
```

```
   Scan the next line of source code into ScanLine;
End While
```

6.3.9. Scanning Algorithm to Create Table PkgClsMtd

The PKGCLSMTD Table is created by the CodeScanner to keep track of all the methods declared in the application.

The **table columns** are Package to store package being scanned, Class to store the class, Method to store the method being scanned, Qualifier to store the method qualifier such as private, public or protected and Return to store the type of value returned by the method.

This table is needed to determine the package and class of a method when we scan a call to a method in another package before the scan of the package has taken place. At that time it is unknown what package or class the method belongs to. We temporarily use the current package and class being scanned to store the method in Table T6_MethodCall and then use the information in this table to update the information in the Table T6_MethodCall. This is done after the entire source is scanned and table PKGCLSMTD is available for reference.

Package	Class	Method	Qualifier	Return
Pkg1	C1	M1	“private”	“void”

Table 9 Table PKGCLSMTD Table

```
// Process when scanning the line of java source code

// Add logic to determine if a class method is being declared

Scan the next line of source code into a variable ScanLine;
```

```
While ScanLine is not a comment or blank line Do
  If a class method is being declared Then
     Pkg = current pkg code being scanned;
     Class = current class code being scanned;
     Method = method being declared in ScanLine;
     Mod = Qualifier of the method being scanned;
     Ret = Type of object returned
     Store above 4 entities in table PkgClsMtd;
     Scan the next line of source code into ScanLine;
   End If
   Continuation Logic….
End While
```

6.3.10. Scanning Algorithm to Create Table PKGIMPORT

The **table columns** are Package to store package being scanned, ImportedPkg to store the package being imported, SourceFile to store name of the file that imports a package, and LineNum to store the line number in the source file where the import statement is encountered.

This table is needed by analyzer when it is determining if package coupling exists between methods in different packages.

Package	ImportedPkg	SourceFile	LineNum
Pkg1	Pkg2	File1	#

Table 10 PKGIMPORT Table

```
// Process when scanning the line of java source code

Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
     Pkg = Current package being scanned;
     Srcfile = Current file being scanned
     If “import” keyword is encountered Then
```

```
            If import of package (based on constructs) Then
                  Pkg2 = Package being imported;
            End If
      End If
      LineNum = Line number of the current ScanLine;
      Store Pkg, Pkg2, Srcfile, LineNum in PKGIMPORT;
      Scan the next line of source code into ScanLine;
End While
```

6.3.11. Scanning Algorithm to Create Extends Table

A table needed to identify parent/child class is an Extends Table.

The **table columns** are Package to store package being scanned, ParentClass to store the class that is being extended, and InheritedClass to store name the current class that extends the ParentClass.

This table is needed by analyzer when it is determining if a class in a package is a parent or a child class. This is used to determine in conjunction with other table information whether a parent or child class changes a secure variable or calls a secure method.

Package	ParentClass	InheritedClass
Pkg1	C1	C2

Table 11 Extends Table

```
// Process when scanning the line of java source code

Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
      Pkg = current pkg code being scanned;
      Class = current class code being scanned;
      Method1 = current method code being scanned;
      LineNum = LineNumber of the current ScanLine;
```

```
/* Subclass identified by a construct “Extends”
      If class2 extends Class Then
            Store Pkg,Class,class2 in Extends Table;
      End
      Scan the next line of source code into ScanLine;
End
```

6.3.12. Scanning Algorithm to Create Table InnerClass

The **table columns** are Package to store package of the secure variable, Class to store the class being scanned, InnerClass to store the innerclass being scanned and LineNum to store the line number of the line being scanned in the source code.

This table is needed as a reference for which innerclass belongs to which class. It is useful later to make the determination of method calling patterns.

Package	Class	InnerClass	LineNum
Pkg1	C1	IC1	#

Table 12 Table InnerClass

```
// Process when scanning the line of java source code

Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
   If inner class is being declared in ScanLine Then
      LineNum = LineNumber of the current ScanLine;
      Pkg = current pkg code being scanned;
      Class = current class code being scanned;
      InnerClass = innerclass code being scanned;
      Store Pkg,Class,InnerClass LineNum in InnerClass;
      Call HandleInnerClass() function;
   End If
   Continuation of Logic …
End While
```

6.3.13. Scanning Algorithm to Create Table SecureClass

This table is created from the T1_SecureVar table by deleting its last column.

The **table columns** are Package to store package of the secure variable, and Class to store the class containing the secure variable.

This table is needed to create an internal list that helps in quick reference check for a secure pkg/class tuple. The information is used by the analyzer in many of the algorithms to identify CR methods.

Table 13 SecureClass

Package	Class
Pkg1	Class1

```
// Process when scanning the SV.txt file
Read the secure variables list file provided by Architect;

While not end of input Do
     Read a line of the input;
     Separate fields and store pkg and class fields into table
SecureClass;
End While
Remove duplicate entries from the table SecureClass
```

6.3.14. Scanning Algorithm to Create Table ObjectMethod

The **table columns** are Package to store package being scanned, Class to store the class , Object to store the object instantiated by main(), Method to store the method/constructor used to create object and LineNum to store the line number in the source file where this code is located.

This table is needed by the analyzer to check the creation of objects of secure class so that they can be recorded and noted in the output to the user.

Table 14 ObjectMethod

Package	Class	Object	Method	LineNum
Pkg1	Class1	Object1	Method1	#

```
// Process when scanning the line of java source code in main() routine

Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
     if new object is created in ScanLine Then
          pkg1 = current package being scanned
          objectclass= Extract class from ScanLine
          objectname= Extract name of object from ScanLine
          constructor= Extract method called from ScanLine
          Store pkg, objectclass, objectname, constructor,
          linecount in ObjectMethod Table
     End If
     Continuation of logic …
     Scan the next line of source code into ScanLine;
End While
```

6.3.15. Scanning Algorithm to Create Table ObjectMethodCall

The **table columns** are Package to store package being scanned, Class to store the class of the object created , Object to store the object instantiated by main(), MethodCall to store the method called by the object and LineNum to store the line number in the source file where this code is located.

This table is needed by the analyzer to check the call to secure methods by the object of secure class or non secure class so that they can be recorded and noted in the output to the user.

Table 15 ObjectMethodCall

Package	Class	Object	MethodCall	LineNum
Pkg1	Class1	Object1	Method1	#

// Process when scanning the line of java source code in main() routine

```
Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
   If  "object.method()" construct is identified in ScanLine Then
     objectname = extract objectname from ScanLine
     methodname = extract attribute from ScanLine
     pkg = pkg currently being scanned
     lnum = line number of the line being scanned
     Get <class> of objectname from ObjectMethod table
     INSERT    pkg,objectname,class,    methodname,lnum    into
     ObjectMethodCallTable
   End IF
   Continuation of logic …
   Scan the next line of source code into ScanLine;
End While
```

6.3.16. Scanning Algorithm to Create Table ObjectCopy

The **table columns** are Package to store package being scanned, Class to store the class of the object created , Object1 to store the object instantiated by main() which copies CopiesObject, CopiedObject2 to store the object being copied into Object1 and LineNum to store the line number in the source file where this code is located.

This table is needed by the analyzer to check the creation of objects of secure class so that they can be recorded and noted in the output to the user.

Table 16 ObjectCopy

Package	Class	Object1	CopiedObject2	LineNum
Pkg1	Class1	Object1	Object2	#

```
// Process when scanning the line of java source code in main() routine

Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
   If  "object1 = object2" construct is identified in ScanLine Then
      object1 = extract objectname1 from ScanLine
      object2 = extract objectname2 from ScanLine
      pkg = pkg currently being scanned
      lnum = line number of the line being scanned
      Get <class> of object1 from ObjectMethod table
      INSERT pkg, class, object1, object2,lnum into ObjectCopy
      Table
   End IF
   Continuation of logic …
   Scan the next line of source code into ScanLine;
End While
```

6.3.17. Scanning Algorithm to Create Table MainClass

The **table columns** are Package to store package being scanned, and Class to store the class in which main() is a declared routine.

This table is created for use in providing ouput to the user about the classes that have declared a main() method.

Table 17 MainClass

Package	Class
Pkg1	Class1

```
// Process when scanning the line of java source code in main() routine

Scan the next line of source code into a variable ScanLine;
While ScanLine is not a comment or blank line Do
   If  main class declaration construct is identified in ScanLine Then
      pkg = pkg currently being scanned
      class = class currently being scanned
      INSERT pkg, class into MainClass Table
   End IF
   Continuation of logic …
   Scan the next line of source code into ScanLine;
End While
```

6.4. Integrity Security Analyzer

This section explains the working of the Integrity Security Analyzer. The Analyzer starts analyzing after the CodeScanner has executed and some of the data in the tables has been set up correctly. The Analyzer starts analyzing for each of the IBC conditions listed in the Section 4 of this document. The following section details the analyzing algorithm for each of the IBCs.

6.4.1. List of distinct Tables Created by the Analyzer

The analyzer creates these additional tables when analyzing the contents of the tables creating during the Code Scanning phase.

1) CRMethod- Contains “pkg, Class, Method, MsgStr, LineNum”

The CR method table is built by the analyzer to store entries for each method in the java code that the analyzer determines to be a CR (Code Review) method. It is possible that entries for the same method may be duplicated because the same method may be a CR method due to multiple IBCs being satisfied by that method.

The **table columns** are Package to store package name, Class to store the class, Method to store the name of CR (Code Review) method, MsgStr to store the message to detail the reason why the method has been identified as a CR method.

This table is used by the ProvideResults() function to display the information to the user.

Table 18 CRMethod Table

Package	Class	Method	MsgStr	LineNum
Pkg1	Class1	Object1	Message String	#

2) UCRMethod - Contains “pkg, Class, Method”

The UCRMethod table is derived from the CRMethod table by deleting the duplicates in the table and deleting the last 2 columns of the CRMethod table.

The **table columns** are Package to store package name, Class to store the class, and Method to store the name of CR method.

This table is used by the ProvideResults() function to display the information to the user.

Table 19 UCRMethod Table

Package	Class	Method
Pkg1	Class1	Object1

3) Dynamic Message - Contains "pkg, Class, Outstr, LineNum"

The Dynamic Message Table is built by the analyzer from the 4 tables ObjectMethod, ObjectAssignment, ObjectMethodCall, and ObjectCopy used to analyze main(). The OutStr provides a meaningful message to the use to understand what is a concern in the main() class.

The **table columns** are Package to store package name, Class to store the class, OutStr to store the message to detail the reason why the main() method construct has been identified to be flagged to the user, and LineNum to store the line number in the source file where the particular construct is encountered.

This table is used by the ProvideResults() function to display the information to the user.

Table 20 DynamicMessage Table

Package	Class	OutStr	LineNum
Pkg1	Class1	Output String1	Line #

6.4.2. Algorithm for Table T3_Var_Modifier

Before listing all the IBCs we ensure that all variables are declared as private or protected.

IBC: *A variable V is defined in a class, and it is not defined as private or protected, then flag the declaration.*

Tables used: T3_Var_Modifier

Brief Algorithm:

Read every row of the table and check the modifier of a variable.

If modifier is not “Private” or “Protected” then raise a flag

Detailed Algorithm:

// Process Table T3_Var_Modifier;

```
K = size of Table T3_Var_Modifier;
I = 1;
For I = 1 : K Do
      Read the List entry at I into pkg, class, var, mod, linenum;
      If List[I].mod is not equal to “Private” or “Protected” Then
            Write (“Error: Variable <var> not declared as private
                  at line number <LineNum>, please fix”);
      End If
      I = I + 1;
End For
```

6.4.3. Algorithm for Direct Change in Secure Class

IBC (Direct Change in Secure Class):

A method M is defined in a secure class, and it changes the value of a secure variable in the class.

Tables used: T1_SecureVar, T4_ChangedVar

Brief Algorithm:

Read every row of the table T4_ChangedVar.

Check if the pkg/class/changedvar entry is in table T1_SecureVar

If yes then store the current method in CRMethod table alongwith a suitable message identifying IBC reason.

Detailed Algorithm:

ProcessTable 4_6_1A()

```
// Process Table T4_ChangedVar

K = size of Table T4_ChangedVar;
For I = 1:K Do
     Read the List entry at I into pkg,class,method,changedvar,
     linenum;
     If pkg/class/changedvar entry in Table T1_SecureVar then
          Outputstr="Warning: Secure variable <changedsecvar>
          changed at line number <linenum>, please review
          <pkg>,<class>,<method>";
          Add pkg/class/method/Outputstr/linenum entry to
CRMethod Table;
     End
End For
```

Table created by the CodeAnalyzer as a result of the processing of Table T4_ChangedVar is the CRMethod

6.4.4. Algorithm for Change to Variable Referred by CR Method in Secure Class

IBC (Change to Variable Referred by CR Method in Secure Class):

A method M is defined in a secure class, and it changes the value of a non-secure variable in the class that is used by a CR method in the class to modify the value of a secure variable.

Tables used: T1_SecureVar, T4_ChangedVar, T5_UsedVar, SecureClass

Detailed Algorithm:

ProcessTable5_1_6_1B()

```
// Process Table T5_UsedVar

I = size of Table T4_ChangedVar;
```

```
J = size of Table T5_UsedVar;
k = 1;
l = 1;
for j = 1 : J
     Read the T5_UsedVar entry at jth row into
     pkg,class,method,usedvar, linenum;
     for i = 1 : I
          Read the T4_ChangedVar entry at ith row into
          PKG,CLS,MTD, CV,LN;
          If PKG/CLS/MTD == a pkg,class,method Then
               If PKG/CLS/CV entry is in Table T1_SecureVar Then
                    Look for PKG/CLS/AnyMTD/CV where CV ==
usedvar
                    If PKG/CLS in SecureClass Table
                         Declare AnyMTD as a CR method;
                         OutputStr="Indirect change to secure
                         variable by <AnyMTD> Method at Line
                         Num <LN> "
                         Add PKG/CLS/AnyMTD/OutputStr/LN to
                         CRMethod table;
                    End If
               End If
          End If
     End For
End For
```

6.4.5. Algorithm for Call CR Method in Secure Class

IBC (Call CR Method in Secure Class):

A method M is defined in a secure class, and it calls a CR method in the class in order to change the value of a secure variable.

Tables used: SecureClass, T6_MethodCall, CRMethod

Detailed Algorithm:

ProcessTable6_1C()

```
// Process Table T6_MethodCall

For I in 1:Size of Table T6_MethodCall Do
     Read          the          Ith          List          into
     pkg,class,method1,pkg2,class2,calledmethod, linenum;
     If pkg/class and pkg2/class2 in SecureClass table Then
          If pkg2/class2/calledmethod is in any row of CRMethod
          Table Col1/2/3 Then
               OutputStr="Warning:  Secure  Method  Call  of
               <calledmethod> method at line number <linenum>,
               please review <pkg>,<class>, <method>");
               Store   Pkg/Class/method1/Outputstr/linenum   in
               CRMethod table;
          End If
     End If
     I = I + 1;
End For
```

6.4.6. Algorithm for Method Called by CR Method in Secure Class

IBC (Method Called by CR Method in Secure Class):

A method M is defined in a secure class, and it is called by a CR method.

Tables used: SecureClass, T6_MethodCall, CRMethod

Detailed Algorithm:

ProcessTable6_6_1D()

```
// Process Table T6_MethodCall

For I in 1 : Size of Table T6_MethodCall Do
     Read          the          Ith          List          into
     pkg,class,method1,pkg2,class2,calledmethod, linenum;
     If pkg/class and pkg2/class2 in SecureClass Then
          If pkg/class/method1 is in any row of CRMethod Table
          Col1/2/3 and calledmethod return type is not void Then
               OutputStr="Warning:    Secure    Method    calls
               <calledmethod> method at line number <linenum>,
               please review <pkg2>,<class2>, <calledmethod>");
```

```
                Store  Pkg2/class2/calledmethod/Outputstr/linenum
                in CRMethod table;
           End If
     End If
End For
```

6.4.7. Algorithm for Call CR Method in Inner Class of Secure Class

IBC (Method in Inner class of Secure Class):

A method M is defined in an inner class of a secure class.

IBC (Call CR Method in Inner Class of Secure Class):

A method M is defined in a secure outer class, and it calls a method in an inner class of the secure outer class.

Tables used: SecureClass, InnerClass, PKGCLSMTD,

Detailed Algorithm:

ProcessRule_6_1E()

// Process Tables for 6-1E

```
K = size of Table Innerclass;
I = 1;
For I = 1: K Do
     Read    the    List    entry    at    Ith    row    into
pkg,class,InnerClass,linenum;
     If pkg/class entry in SecureClass Table Then
          Look for pkg2, class2, method2 from PkgClsMtd WHERE
pkg2=pkg AND
          class2=Innerclass
          If found Then
               OutputStr="Warning:  Innerclass  <Innerclass>  of
               secure class
```

```
                <class>, Please review pkg <pkg2>,class <class>,
                innerclass
                <innerclass>, innerclass method <method2>”);
                Store   pkg/class/ICMethod/OutputStr/linenum   in
                CRMethod Table;
           End If
      End If
End For
```

6.4.8. Algorithm for Change to Variable Referred by Inner Class of Secure Class

IBC (Change to Variable Referred by Inner Class of Secure Class):

A method M is defined in a secure outer class, and it changes the value of a variable in the class that is referred by a method in an inner class of the secure outer class.

Tables used: T5_UsedVar, T4_ChangedVar, T1_SecureVar, SecureClass, Extends,

Detailed Algorithm:

ProcessTable5_20_6_1F()

```
for I= 1: Sizeof T5_UsedVar:
      Get data into pkg, myclass, method, usedvar, linenum
      for J= 1: Size of T4_ChangedVar:
            Get data intp PKG, CLS, MTD, CV, LN
             if PKG==pkg and CLS==myclass and MTD==method:
                  for K =1 : sizeof T1_SecureVar
                        Get data into pkgt1, classt1, svt1
                        if [PKG,CLS] in secureclasses:
                              if  PKG==pkgt1  and  CLS==classt1  and
                              CV==svt1     and     pkg==PKG     and
                              myclass==CLS:
```

```
                                    SELECT * FROM T4_ChangedVar
                                    where ChangedVar = usedvar into
                                    pkg4,cls4,ANYMTD,ln4
                                    for each record in T4_ChangedVar
DO
                                         SELECT * FROM EXTENDS into
                                         pkgext,        parentclass,
                                         childclass
                                         For each record in EXTENDS
                                              if  cls4==myclass  or
                                         (cls4==parentclass      and
                                         childclass==myclass):
                                                   Outputstr=("
                                                   Warning:    6-1F
                                                   Possible
                                                   indirect  change
                                                   to        Secure
                                                   variable <CV> by
                                                   method <ANYMTD>,
                                                   please    review
                                                   pkg       <pkg>,
                                                   secure     class
                                                   <myclass>,
                                                   method  <ANYMTD>
                                                   at  line  number
                                                   <ln4>

                                                   INSERT        OR
                                                   REPLACE     INTO
                                                   CRMethod  VALUES
                                                   pkg4,      cls4,
                                                   ANYMTD,Outputstr
                                                   ,ln4
                                              End If
                                         End For
                                    End For
                               End If
                          End If
                     End For
                End If
```

```
      End For
End For
return()
```

Tables used: T7_InnClsChgdVar, T1_SecureVar

Detailed Algorithm:

ProcessTable7()

// Process Table T7_InnClsChgdVar

```
I = size of Table T1_SecureVar;
J = size of Table T7_InnClsChgdVar;
for j = 1 : J
      Read the T7_InnClsChgdVar entry at jth row into
      pkg,class,innerclass,icmethod, changedvar,linenum;
      for i = 1 : I
            Read the T1_SecureVar entry at ith row into
PKG,CLS,SecVar;
            If PKG/CLS/SecVar == a pkg,class,changedvar Then
                  Declare icmethod as a CR method;
                  OutputStr="Secure variable changed by innerclass
                  method <icmethod> at Line Num <linenum> "
                  Add pkg/innerclass/icemthod/OutputStr/linenum to
            CRMethod
            End If
      End For
End For
```

Tables used: T7_InnClsChgdVar, T8_InnClsUsedVar, T1_SecureVar, SecureClass

Detailed Algorithm:

ProcessTable8_1_6_1F()

// Process Table T8_InnClsUsedVar

```
J = size of Table T8_InnClsUsedVar;
K = size of Table T7_InnClsChgdVar
for j = 1 : J
```

```
     Read the T8_InnClsUsedVar entry at jth row into
     pkg,class,ic,icmethod,usedvar,
     linenum;
     for i = 1 : K
          Read the T7_InnClsChgdVar entry at ith row into
          PKG,CLS,ICLS,ICMTD,CV,LN;
          If PKG/CLS/ICLS/ICMTD == a pkg,class,ic,icmethod Then
               If PKG/CLS/CV entry is in Table T1_SecureVar Then
                    Look for PKG/CLS/AnyMTD/CV where CV ==
usedvar
                    If PKG/CLS in SecureClass Table
                         Declare AnyMTD as a CR method;
                         OutputStr="Indirect change to secure
                         variable by <AnyMTD> Method at Line
                         Num <LN> "
                         Add PKG/CLS/AnyMTD/OutputStr/LN to
                         CRMethod table;
                    End If
               End If
          End If
     End For
End For
```

6.4.9. Algorithm for Method Called by Inner Class of Secure Class

IBC (Method Called by Inner Class of Secure Class):

A method M is defined in a secure outer class, and it is called by a CR method in an inner class of the secure outer class.

Tables used: InnerClass, T6_MethodCall, SecureClass

Detailed Algorithm:

ProcessTable6_6_1G()

```
// Process Table T6_MethodCall
K = size of Table T6_MethodCall;
```

```
I = 1;
For I = 1:K Do
     Read the Ith List into pkg,class,method1,pkg2,class2,
     calledmethod, linenum;
     If pkg/class in any row of InnerClass Table Col1/Col3 Then
          If pkg=pkg2 and pkg/class in SecureClass Then
               OutputStr="Warning: Secure Inner Class Method
               <method1> calls <calledmethod> method at line
               number <linenum>, please review <pkg2>,<class2>,
               <calledmethod>");
               Store Pkg2/class2/calledmethod/Outputstr/linenum
               in CRMethod table;
          End If
     End If
     I = I + 1;
End For
```

6.4.10. **Algorithm for Direct Change to Data in SDWC**

IBC (Change to Secure Data in SDWC):

A method M is defined in a SDWC and it changes the secure data value stored in a database.

Tables used: T4_ChangedVar, T1_SecureVar

Detailed Algorithm:

ProcessTable4_6_1A()

```
// Process Table T4_ChangedVar
K = size of Table T4_ChangedVar;
For I = 1:K Do
     Read the List entry at I into
pkg,class,method,TABLE,linenum;
     If pkg/class/TABLE entry in Table T1_SecureVar then
```

```
            Outputstr="Warning: Secure TABLE <TABLE> changed at
            line number <linenum>, please review
            <pkg>,<class>,<method>";
            Add pkg/class/method/Outputstr/linenum entry to
CRMethod Table;
      End
End For
```

6.4.11. **Algorithm for Call CR Method in SDWC or NSDWC**

IBC (Call CR Method in SDWC or NSDWC):

A method is defined in a SDWC or NSDWC and it calls a CR method in the class.

Tables used: T6_MethodCall, CRMethod

Detailed Algorithm:

ProcessTable6_1C()

```
// Process Table T6_MethodCall

K = size of Table T6_MethodCall;
I = 1;
For I = 1:K Do
      Read the Ith List into
      pkg,class,method1,pkg2,class2,calledmethod, linenum;
      If pkg2/class2/calledmethod in any row of CRMethod Table
      Col1/2/3 Then
            OutputStr="Warning: Secure Method Call of
            <calledmethod> method at line number <linenum>,please
            review <pkg>,<class>, <method>");
            Store Pkg/Class/method1/Outputstr/linenum in CRMethod
            table;
      End If
End For
```

6.4.12. Algorithm for Method Called by CR Method in SDWC or NSDWC

IBC (Method Called by CR Method in SDWC or NSDWC):

A method M is defined in a SDWC or NSDWC and it is called by a CR method in the class.

Tables used: T6_MethodCall, CRMethod

Detailed Algorithm:

ProcessTable6_6_1D()

```
// Process Table T6_MethodCall

K = size of Table T6_MethodCall;
I = 1;
For I=1:K Do
     Read the Ith List into
     pkg,class,method1,pkg2,class2,calledmethod, linenum;
     If pkg/class/method1 is in any row of CRMethod Table
     Col1/2/3 and calledmethod return type is not void Then
          OutputStr="Warning: Secure Method <method1> calls
          <calledmethod> method at line number <linenum>, please
          review <pkg2>,<class2>, <calledmethod>");
          Store Pkg2/Class2/calledmethod/Outputstr/linenum in
          CRMethod;
     End If
End For
```

6.4.13. Algorithm for Call CR Method in other classes

IBC (Call CR Method in other classes):

A method M is defined in a class and it calls a CR method defined in another class.

Tables used: T6_MethodCall, CRMethod

Detailed Algorithm:

ProcessTable6_3A()

```
// Process Table T6_MethodCall

For I in 1:Size of Table T6_MethodCall Do
     Read          the          Ith          List          into
     pkg,class,method1,pkg2,class2,calledmethod, linenum;
     If pkg = pkg2 and class != class2 Then
          If pkg2/class2/calledmethod is in any row of CRMethod
          Table Col1/2/3 Then
               OutputStr="Warning:   Secure   Method   Call   of
               <calledmethod> method at line number <linenum>,
               please review <pkg>,<class>, <method1>");
               Store   Pkg/Class/method1/Outputstr/linenum   in
               CRMethod table;
          End If
     End If
     I = I + 1;
End For
```

6.4.14. Algorithm for Method Called by CR Method in other class

IBC (Method Called by CR Method in other class)

A method M is defined in a class and it is called by a CR method in another class.

Tables used: T6_MethodCall, CRMethod

Detailed Algorithm:

ProcessTable6_3B()

```
// Process Table T6_MethodCall

For I in 1:Size of Table T6_MethodCall Do
     Read          the          Ith          List          into
     pkg,class,method1,pkg2,class2,calledmethod, linenum;
     If pkg = pkg2 and class != class2 Then
```

```
            If pkg/class/method1 is in any row of CRMethod Table
            Col1/2/3 and calledmethod return type is not void Then
                  OutputStr="Warning: Secure Method <method1> calls
                  <calledmethod> method at line number <linenum>,
                  please review <pkg2>,<class2>, <calledmethod>");
                  Store  pkg2/class2/calledmethod/Outputstr/linenum
                  in CRMethod table;
            End If
      End If
      I = I + 1;
End For
```

6.4.15. Algorithm for Change Variable Referred by CR Method in NSC

*IBC (**Change Variable Referred by CR Method in NSC**):*

A method M is defined in a NSC and it changes the value of a variable referred by a CR method in the class.

Tables used: T5_UsedVar, T4_ChangedVar, T1_SecureVar, SecureClass

Detailed Algorithm:

ProcessTable5_2_for_6_3C()
Note: ProcessTable8_2_for_6_3C() is similar to this process except it works on Table T8_InnClsUsedVar

```
// Process Table T5_UsedVar

I = size of Table T4_ChangedVar;
J = size of Table T5_UsedVar;
for j = 1 : J
      Read the T5_UsedVar entry at jth row into pkg,class,method,
      usedvar,linenum;
      for i = 1 : I
            Read  the  T4_ChangedVar  entry  at  ith  row  into
PKG,CLS,MTD,          CV,LN;
            If PKG/CLS/MTD == a pkg,class,method Then
                  If PKG/CLS/CV entry is in Table T1_SecureVar Then
```

```
                    Look for PKG/CLS/AnyMTD/CV where CV ==
usedvar
                    If PKG/CLS Not in SecureClass Table
                         Declare AnyMTD as a CR method;
                         OutputStr="Indirect change to variable
                         by <AnyMTD> Method at Line Num <LN> "
                         Add PKG/CLS/AnyMTD/OutputStr/LN to
                         CRMethod table;
                    End If
               End If
          End If
     End For
End For
```

6.4.16. Algorithm for Call CR method in NSC

IBC (Call CR method in NSC):

A method M is defined in a NSC and it calls a CR method in the class.

Tables used: T6_MethodCall, CRMethod, SecureClass

Detailed Algorithm:

ProcessTable6_3D()

```
// Process Table T6_MethodCall

For I in 1:Size of Table T6_MethodCall Do
     Read the Ith List into
     pkg,class,method1,pkg2,class2,calledmethod, linenum;
     If pkg = pkg2 and class = class2 AND pkg/class Not in
     SecureClass Then
          If pkg2/class2/calledmethod is in any row of CRMethod
          Table Col1/2/3 Then
               OutputStr="Warning: Secure Method Call of
               <calledmethod> method at line number <linenum>,
               please review <pkg>,<class>, <method1>");
               Store Pkg/Class/method1/Outputstr/linenum in
               CRMethod table;
          End If
```

```
      End If
      I = I + 1;
End For
```

6.4.17. Algorithm for Method Called by a CR method in NSC

IBC (Method Called by a CR method in NSC):

A method M is defined in a NSC and it is called by a CR method in the class.

Tables used: T6_MethodCall, CRMethod, SecureClass

Detailed Algorithm:

ProcessTable6_3E() in same class and ProcessTable6_6_3E() for other class

```
// Process Table T6_MethodCall

For I in 1:Size of Table T6_MethodCall Do
      Read          the          Ith          List          into
      pkg,class,method1,pkg2,class2,calledmethod, linenum;
      If pkg = pkg2 and class = class2 and pkg/class Not in
      SecureClass Then
            If pkg/class/method1 is in any row of CRMethod Table
            Col1/2/3 and calledmethod return type is not void Then
                  OutputStr="Warning: Secure Method <method1> calls
                  <calledmethod> method at line number <linenum>,
                  please review <pkg2>,<class2>, <calledmethod>");
                  Store  pkg2/class2/calledmethod/Outputstr/linenum
                  in CRMethod table;
            End If
      End If
      I = I + 1;
End For
```

6.4.18. Algorithm for Change Protected Secure Variable in Parent

IBC (Change Protected Secure Variable in Parent):

A method M is defined in a class, and it changes the value of a protected secure variable (SV) in its parent class (e.g., pSecVar2 in Fig. X).

Tables used: T4_ChangedVar, EXTENDS, T1_SecureVar, T3_Var_Modifier

Detailed Algorithm:

ProcessTable4_6_4A_B()

```
for I =1: size of Table T4_ChangedVar do
      Read the Ith entry into pkg, myclass, method, changedvar,
      linenum
      for J =1 : size of EXTENDS Table do
            Read the Jth row into pkgext, parentclass, childclass
            if pkgext==pkg and childclass==myclass THEN
            for K = 1: Size of Table T1_SecureVar do
                  Read Kth row into pkg1, myclass1, changedvar1
                  IF   pkg==pkg1   and   parentclass==myclass1   and
                  changedvar==changedvar1                        and
                  [pkg,parentclass,changedvar,  'protected']  tuple
                  in T3_Var_ModifierTableItemsList THEN
                        Outputstr=(" Warning: 6-4A Protected Secure
                        variable <changedvar>  changed by subclass
                        <myclass>,  please  review  new  CR  method
                        <pkg> , class <myclass> and method <method>
                        at line number <linenum>")
                        INSERT      INTO      CRMethod      VALUES
                        (pkg,myclass,method,Outputstr,linenum))
                  END IF
            END for
      END for
END For
```

6.4.19. Algorithm for Change Protected Non-secure Variable in Parent

IBC (Change Protected Non-secure Variable in Parent):

A method M is defined in a class and it changes the value of a protected NSV in its parent class and any CR method in the parent class uses the protected NSV

Tables used: T4_ChangedVar, EXTENDS, T1_SecureVar, T3_Var_Modifier

Detailed Algorithm:

ProcessTable4_6_4A_B()

```
for I =1: size of Table T4_ChangedVar do
      Read the Ith entry into pkg, myclass, method, changedvar,
      linenum
      for J =1 : size of EXTENDS Table do
            Read the Jth row into pkgext, parentclass, childclass
            if pkgext==pkg and childclass==myclass THEN
            for K = 1: Size of Table T1_SecureVar do
                  Read Kth row into pkg1, myclass1, changedvar1
                  IF   pkg==pkg1   and   parentclass==myclass1   and
                  changedvar!=changedvar1                        and
                  ([pkg,parentclass,changedvar,   'protected']   in
                  T3_Var_ModifierTableItemsList) THEN
                        Outputstr=(" Warning: 6-4B Protected Non-
                        Secure variable <changedvar>   of secure
                        class changed by subclass< myclass>, please
                        review new CR method pkg <pkg>,class
                        <myclass>, and method <method> at line
                        number <linenum>"
                        INSERT OR REPLACE INTO CRMethod VALUES
                        (pkg,myclass,method,Outputstr,linenum)
                  END IF
            END for
      END for
END For
```

6.4.20. Algorithm for Access Protected Member (Variable and Method) from other classes

IBC (Access Protected Member from Other Classes):

A method M is defined in a class, and the class is not a child class, and the method M accesses a protected member (e.g., protected variable or method) in a parent class.

This IBC includes a method M in a child class, which accesses a protected variable or method that is not defined in its parent class.

Tables used: T5_UsedVar, EXTENDS, SecureClass, T3_Var_Modifier

Detailed Algorithm:

ProcessVariableRule6_4C_Use()

```
for I = 1: Size of Table T5_UsedVar
     Read contents of row I into pkg, myclass, method, usedvar,
     linenum
     for J = 1: Size of Table T3_Var_Modifier
          Read contents of row J into pkg1, myclass1, var1,
          modifier, lnum
          if var1==usedvar and pkg1==pkg and
     modifier=='protected':
               for K = 1: Sizeof EXTENDS table
                    Read contents of row K into
                    pkgext,parentclass, childclass
                    IF [pkg1,parentclass] in secureclasses
               THEN:
                         IF pkgext==pkg1 and
                         parentclass==myclass1 and
                         childclass!=myclass and parentclass !=
                         myclass THEN
                              Outputstr=(" Warning: 6-4C1
                              Protected variable <usedvar> in
                              Superclass <parentclass>
                              accessed/used by class
                              <myclass>, please review new CR
                              method pkg <pkg>,class
                              <myclass>, and method <method>
                              at line number <linenum>")
                              INSERT OR REPLACE INTO CRMethod
                              VALUES
                              (pkg,myclass,method,Outputstr,li
                              nenum)
```

```
                              END IF
                         END IF
                    END For
               END IF
          END For
END for
```

Tables used: T4_ChangedVar, EXTENDS, SecureClass, T3_Var_Modifier

Detailed Algorithm:

ProcessVariableRule6_4C_Change()

```
for I = 1: Size of Table T4_ChangedVar
     Read contents of row I into pkg, myclass, method,
     changedvar, linenum
     for J = 1: Size of Table T3_Var_Modifier
          Read contents of row J into pkg1, myclass1, var1,
          modifier, lnum
          if     var1==changedvar     and     pkg1==pkg     and
          modifier=='protected':
               for K = 1: Sizeof EXTENDS table
                    Read contents of row K into pkgext,
                    parentclass, childclass
                    IF [pkg1,parentclass] in secureclasses
               THEN:
                         IF          pkgext==pkg1          and
                         parentclass==myclass1             and
                         childclass!=myclass and parentclass !=
                         myclass THEN
                              Outputstr=("   Warning:   6-4C2
                              Protected variable <changedvar>
                              in   Superclass   <parentclass>
                              accessed/used      by      class
                              <myclass>, please review new CR
                              method      pkg      <pkg>,class
                              <myclass>, and method <method>
                              at line number <linenum>")
```

```
                                    INSERT OR REPLACE INTO CRMethod
                                    VALUES
                                    (pkg,myclass,method,Outputstr,li
                                    nenum)
                               END IF
                          END IF
                     END For
               END IF
          END For
END for
```

Additionally ProcessMethods6_4C() also process contents of Table T6_MethodCall for this clause.

6.4.21. Algorithm for Call CR Method in another package

IBC (Call CR Method in another package):

there is a package import relationship between two packages, and a method M is defined in a class in a package, and it calls a CR method in a class in another package.

Tables used: T6_MethodCall, PKGIMPORT, CRMethod

Detailed Algorithm:

ProcessTable6_6_5A()

```
// Process Table T6_MethodCall

K = size of Table T6_MethodCall;
I = 1;
For I = 1:K Do
     Read          the          Ith          List          into
     pkg1,class1,method1,pkg2,class2,calledmethod, linenum;
     If pkg1/pkg2 is in PKGIMPORT table AND pkg1 Not = pkg2 THEN
          If pkg2/class2/calledmethod is in any row of CRMethod
          Table Col1/2/3 Then
```

```
                OutputStr="Warning: Secure Method Call of
                <pkg2>/<class2>/<calledmethod> method by
                <pkg1>/<class1>/method1 at line number <linenum>,
                please review both these methods");
                Store Pkg1/Class1/method1/Outputstr/linenum in
                CRMethod table;
          End If
     End If
End For
```

6.4.22. Algorithm for Called by CR Method in another package

IBC (Called by CR Method in another package):

There is a package import relationship between two packages, and a method M is defined in a class in a package, and it is called by a CR method in a class in another package.

Tables used: T6_MethodCall, PKGIMPORT, CRMethod

Detailed Algorithm:

ProcessTable6_6_5B()

```
// Process Table T6_MethodCall

K = size of Table T6_MethodCall;
I = 1;
For I = 1:K Do
     Read the Ith List into
     pkg1,class1,method1,pkg2,class2,calledmethod, linenum;
     If pkg1/pkg2 is in PKGIMPORT table AND pkg1 Not = pkg2 THEN
          If pkg1/class1/method1 is in any row of CRMethod Table
          Col1/2/3 THEN
               OutputStr="Warning: Secure
               <pkg1>/<class1>/<method1> Method Calls Non Secure
               method of <pkg2>/<class2>/<calledmethod> method
               at line number <linenum>, review the called
               method");
```

```
                Store  pkg2/class2/calledmethod/Outputstr/linenum
                in CRMethod table;
            End If
      End If
End For
```

6.4.23. Algorithm for main () and dynamic analysis Dynamic Rule 1_1 and 1_2

Data Structures/Tables needed by the analyzer for main() analysis are MainClass, ObjectMethod, ObjectCopy and ObjectMethodCall, DynamicMessage

Rule 1.1 If the class of a new object is secure class, alert to the creation of a secure object

Rule 1.2 If the class of new object is secure class, flag object creation using a constructor if constructor is a CR method

Tables used: ObjectMethod, SecureClass, UCRMethod

Detailed Algorithm:

ProcessDynamicRule1()

```
K = Size of Table ObjectMethod
L = Size of Table UCRMethod
For I = 1:K Do
   Select      Ith      row      FROM      ObjectMethod      into
pkg1,class1,object1,mtd1,ln
   if [pkg1,class1] in list of secureclasses Then
     Outputstr="Warning:  Dynamic  Rule1  -  New  secure  object
     <object1> of secure class <class1> created at line number
     <ln>
     INSERT into DynamicMessage Table pkg1,class1,Outputstr,ln
   For J = 1: L do
     SELECT pkgcr, classcr, methodcr FROM UCRMethod Table
```

```
      IF pkg1 = pkgcr AND class1 = classcr AND mtd1 = methodcr
      Then
           Outputstr=" Warning: Dynamic Rule1_2 - New secure
           object <object1> of class <class1> calls constructor
           <mtd1> at line number <ln>"
           INSERT INTO DynamicMessage pkg1,class1,Outputstr,ln
      End IF
   End For
End For
```

6.4.24. Algorithm for Analyzing main() Dynamic Rule 2

Flag all "object.method()" constructs where method is a CR method

Tables used: ObjectMethodCall, UCRMethod

Detailed Algorithm:

ProcessDynamicRule2()

```
// Process Table ObjectMethodCall

K = size of Table ObjectMethodCall;
For I = 1:K Do
   SELECT * FROM ObjectMethodCall into pkg1,class1,object1,
   calledmethod,lnum
   IF [pkg1,class1,calledmethod] tuple is in Table UCRMethod THEN
     Outputstr="Warning: Dynamic Rule3 - Object <object1> of
     class <class1> calls secure method <calledmethod> at line
     number <lnum>
     INSERT INTO DynamicMessage Table pkg1,class1, Outputstr,lnum
   End IF
End For
```

6.4.25. Algorithm for Analyzing main() Dynamic Rule 3

If secure object is created and then copied (aliased) into another secure object, then flag the aliasing of the object.

Tables used: ObjectCopy, UCRMethod

Detailed Algorithm:

ProcessDynamicRule3()

```
// Process Table ObjectCopy

K = size of Table ObjectCopy;
For I = 1:K Do
   SELECT * FROM ObjectCopy into pkg1,class1,object1,object2,lnum
   IF [pkg1,class1] tuple is in Table SecureClass THEN
     Outputstr="Warning: Dynamic Rule4 - Secure object <object1>
     of secure class <class1> is aliased at line number <lnum>
     INSERT INTO DynamicMessage Table pkg1,class1, Outputstr,lnum
   End IF
End For
```

6.4.26. Algorithm for ProcessDynamicResults()

The DynamicMessage table is created by analyzing routines in main() listed above in the main() analysis. The final processing of the DynamicMessage table is to extract all rows of the table and display them in that order. Each message is a message about integrity security violation in the main modules of the application.

Tables used: Dynamic Message, MainClass

Detailed Algorithm:

```
// Process Table DynamicMessage

K = size of Table DynamicMessage;
For I = 1:K Do
   SELECT * FROM Dynamic Message into mypkg, myclass, outstr,
linenum
   For J = 1: Size of MainClass Do
     SELECT * from MainClass into Pkg2, mymainclass WHERE Pkg2 =
     mypkg
     Send the information to the Review_Out() method for display
     to the User
```

```
End For
```

6.4.27. Overall Code Analysis Algorithm

The Code analysis algorithm is shown along with the Code Scanning algorithm in the previous section. It is easier to understand the algorithm when seen in the context of how the data structures needed for that scanning algorithm are built. All of the scanning algorithms are contained in functions. The overall Code Analysis algorithm calls each of the individual Code Analysis functions described in 9.10 as per the following. The algorithm continues processing the static conditions in a cycle until no more CR methods are discovered. Once the static conditions are processed to completion, then the Dynamic conditions are processed once and the algorithm exits. During this processing CR methods are created and added to the CRMethods table.

Code Analyzer Algorithm

Inputs : None

Outputs: None

```
NumCRBefore=0
NumCRAfter=1
while NumCRAfter  >  NumCRBefore:
      Repeat 2 times steps 2 through 26
      1     NumCRBefore = NumCRAfter
      2     ProcessResultsTable6()
      3     ReviewTable6contents()
      4     ProcessTable4_6_1A()
      5     ProcessTable5_1_6_1B()
      6     ProcessTable6_1C()
      7     ProcessTable6_6_1D()
      8     ProcessRule_6_1E()
      9     ProcessTable5_20_6_1F()
      10    ProcessTable8_1_6_1F()
      11    ProcessTable7()
      12    ProcessTable6_6_1G()
```

```
      13    ProcessTable6_3A()
      14    ProcessTable6_3B()
      15    ProcessTable8_2_for_6_3C()
      16    ProcessTable5_2_for_6_3C()
      17    ProcessTable6_3D()
      18    ProcessTable6_3E()
      19    ProcessTable6_6_3E()
      20    ProcessTable4_6_4A_B()
      21    ProcessVariableRule6_4C_Use()
      22    ProcessVariableRule6_4C_Change()
      23    ProcessMethod6_4C()
      24    ProcessTable6_6_5A()
      25    ProcessTable6_6_5B()
      26    ProcessResultsTable6()
      27    RemoveDuplicatesCRMethod()
      NumCRAfter = NumCRMethods() {Get this from UniqueCRMethod
Table}
28    ProcessDynamicRule1()
29    ProcessDynamicRule2()
30    ProcessDynamicRule3()
```

6.5. Provide Results Algorithm

The algorithm to display the results is shown below along with comments to indicate the nature of the processing.

ProvideResults()

Inputs: None

Outputs: None

```
ProcessResultsTable6()     //Process to discover any more CR
methods
RemoveDBduplicates()       //Remove duplicates in Database tables
ProcessResultsTable6()     //Process once more the first step
RemoveDBduplicates()       //Remove duplicates if any
showMethods_CRM_contents() //Write formatted contents of CRMethod
```

```
                          //table including duplicates to output
                          //file
ProcessResultsTable3()    //Process Table 3 indicating locations
                          //where secure variables are declared
                          //in the Java application
ProcessResultsCRMethod()  //Write entire content of CRMethod
                          table //to the output file
RemoveDuplicatesCRMethod() //Remove duplicates from the CRMethod
                          //table and store in UCRMethod table
ProcessResultsUCRMethod()  //Write out UCRMethod table contents
to
                          //Output file
ProcessResultsDynamic()   //Write contents of Dynamic Message
table
                          //to the output file
```

6.6. Tool Implementation

The CAIS tool is implemented in the Python programming language. Python is an open source programming language that is available for free at the following URL https://www.python.org/

The version used to create the tool is the Python 3.4.2. Python is a self-contained programming language in that most of the open source utilities and libraries are available to be downloaded into the python installation. This version of Python also comes with the native database SQLite3. This serves the purpose of the tool very well since there are no additional set up or installation requirements. The choice of python for the CAIS tool is derived for the ease of the availability of tools/support for the language as well as the ease of use and suitability of python for use in a tool such as this. Python is lightweight and provides an excellent Integrated Development Environment (IDE) for program/script development.

6.6.1. Assumption

It would be difficult to write a scanner that can scan every possible java code structure. Nothing short of a Java compiler would be required to achieve that.

It is important to specify certain coding practices and mention limitations of the tool. When the capability of the tool is enhanced in the future, we can review the coding guidelines. These need to be strictly followed at this time to take maximum advantage of the CAIS capability and reduce errors. This section specifies some **Java coding guidelines** based on the tool limitation. An effort has been made to have minimal constraints, but it is important to follow these for correct use of the CAIS tool.

6.6.1.1. File Organization

Each Java source file should contain a single public class. Each class must be placed in a separate java source file. This applies to non-public classes too.

6.6.1.2. Package Organization

Java classes should be packaged in a new java package for each self-contained project or group of related functionality. Ensure that all packages are in one folder and not scattered over different folders.

6.6.1.3. Source code style guidelines

i) **Comments:** All comment lines must begin with //

Example1

```
// This is a comment in the .java file
```

ii) All comments should be located on a separate line

Example2

```
}
// End of Method
```

Instead of

```
} // End of Method
```

iii) Start and end of blocks of statement

A statement block must starts with { on a separate line

Example 1

```
public void samplemethod
{
```

Instead of

```
Public void samplemethod {
```

Example 2

```
return (0);
}
```

Instead of

```
return (0); }
```

The same brace pattern should also be adopted for try/catch, loops, conditional and switch statements. All try/catch, loops and conditional constructs define a block of code even for single lines of code.

iv) Wrapping Lines

Line wrap is when a statement on a line is wrapped to a new line because it cannot be completed on the single line. Currently the tool has not been designed to handle line wrap. Must write statements that fit on a single line or break long statements into multiple small statements.

v) White Space

Blank Lines: Blank lines improve readability and are handled by the tool. Use a blank line:

- Between sections of a source file
- Between class and interface definitions
- Between methods.
- Between the local variables in a method and its first statement.
- Before a block or single-line comment.
- Between logical sections inside a method to improve readability.
- Before and after comments as applicable

vi) Blank Spaces

Blank spaces should be used in the following circumstances:

- Separate words, operands and operators in a statement with a blank space

  Example:

  ```
  New_Sum = New_Sum + current_value ;
  ```

- A blank space should appear after commas in argument lists.

Example:

```
Call_function ( argument1,<blankspace> argument2)
```

- All binary operators except a period '.' should be separated from operands by spaces.

vii) Declarations

One declaration per line is recommended since it encourages commenting and enhances the clarity of code. The order and position of declaration should be as follows:

- First the static/class variables should be placed in the sequence: First protected class variables, then the private followed by Static fields which should be explicitly instantiated by use of static initializers.
- For all classes current support is for class variables. Declare class variables and use them in the methods. Currently local method variables are not supported by the CAIS tool. If local variables are declared and used for malicious purpose then this occurrence will not be captured by the tool.
- Next the class constructors should be declared.
- This should be followed by the inner classes, if applicable
- Local declarations are supported in Subclasses and innerclasses. Local declaration that hide declarations at higher levels must be avoided. For example, same variable name in an inner block must be avoided.

viii) Standards for Statements

Each line should contain at most one statement. Compound statements are lists of statements enclosed in braces. Braces should be on a separate line by themselves as mentioned earlier.

ix) Standards for Methods

- Use Getters and Setters wherever possible for accessing values of fields.
- Cascading method calls like `method1().method2()` must be avoided as they will not be handled
- Return statements should not include calls to methods

  For example,

```
return 0; //This is OK
return UserChoice;  //This is OK
```

  But …

```
return(Customer.ChangeBalance(Amount));   //This   is   not
supported
```

x) Naming Convention standards

Do not use same names for a class or local variable, argument, or methods as that of other variables, methods in the same package or other packages. Use unique names. Otherwise the tool will not handle this duplication.

xi) Variable Assignments

- Assigning several variables to the same value in a single statement should be avoided, i.e., we should avoid constructs like `var1 = var2 = var3 = 0;`

- Have each simple statement on a separate line.

xii) Documenting & Declaring a Variable

- Declare one variable per line of code.
- If required, document variables with a comment on a new line
- Use different variable names for all the variables used in the program

However the following should be taken into consideration while using variables:

- Instance variables should not be declared public.
- Use of static must be avoided as much as possible since they act like globals.
- Same names for variables or methods must be avoided in subclasses/inner classes.

xiii) Current Tool limitations

The current tool is basic and addresses all typical uses of basic Java language. Advanced Java constructs may not be handled at this time since Java is a large language that supports a lot of add on tools and capabilities. Future versions of the tool could be modified to address some of these limitations.

Here is a list of things not handled by the tool currently:

1) No loops inside of class constructors
2) Nesting of loops beyond 2 levels in any method
3) Nesting of loop with conditional or switch statements beyond 2 levels in any method
4) Complex calls to methods such as `object.method().object.method()`
5) Multi-threaded application

6) Concurrent or parallel programming
7) Network programming
8) Client and Server architecture programming
9) Graphic user Interface programming
10) Web programming
11) while, for, switch, do, if, elseif, else statements should not be in a constructor
12) Local variables

All of these capabilities will require resources to develop these features and those are a team effort that must be employed to take the tool to the next level of capability.

6.6.2. Implementation details

The CAIS tool details are mentioned in this section. The 4 parts of the implementation are 1) SV.txt file which contains the list of secure variables/tables provided by the Architect or Software Engineer. The contents of this file are a list of "pkg class SecureVariable" or a "pkg class SecureTable" tuple on each line of the SV.txt file

Example SV.txt file:

```
=========================================================
Pkg1  class1     Account
Pkg1  class4     account_table
=========================================================
```

The CAIS tool will use this as an input to the CodeScanner. The CodeScanner also writes output to the "OutputReview_backup" file which is a text file that records scanning

activity as it scans packages and classes. After the Analyzer completes processing, the results are provided to the user through another output file "OutputReview". This file records a list of all CR methods and details of the reason why each method is considered a CR method. The OutputReview is also populated by the contents of the UCRMethod table or the Unique CR methods. These are a more concise list after removing all the duplicates from the CRMethod table. A couple of verbose columns are also removed to make the output easy to read. Finally the OutputReview file will also contain an output that shows the different main() methods in the packages and the java code constructs which would be considered as security hotspots at run time when main() executes.

Another file that is created is the "SCRModular.Log" file. This file logs messages from the CodeScanner(), the CodeAnalyzer() and the ProvideResults() routines and all of the routines that are called by main() and these above mentioned routines. The log file logs all debug and execution messages. It is a useful tool when testing and debugging the python source code for the CAIS tool. It is a useful tool for future enhancement of features. It is nearly indispensable due to the nature of this tool.

A word of caution must be mentioned here. The log file is deleted and recreated for every run of the CAIS tool. Depending upon how many files and packages are scanned and how many CR methods are revealed, the log file tends to grow in size quickly and sizes of over 500 MB are not uncommon. The better the java application is written (less coupling and less security hot spots), the lower will be the size of the logfile.

6.7. Tool Performance

The CAIS tool has been tested with the Unit test cases and with the entire Java application being tested together for performance, both qualitative and quantitative. The testing was carried out by using a test case for each of the different IB conditions that have been listed in the Section 5 of this dissertation.

These 24 test cases reside in 34 different java source code files. There are 236 methods in 34 different classes that reside in 4 different packages. A total of 2870 lines of code were scanned and analyzed. The number of malicious conditions is varied by using the "Secure Variable file" listing package, class, variable. The number of malicious conditions (A secure variable addition causes one or more malicious conditions to be added) were varied between 1 and 55. 10 secure tables were also used as a part of the 55 secure data.

Based on these test conditions the tool successfully identifies all of the CR methods in the application. The tool is designed to be configurable. The expertise needed is to be able to understand the implementation so that modifications can be done to handle new usages and constructs in the application program. It is out of the scope of this dissertation to include all of the typical uses of the Java programming language.

The performance of the tool with respect to the test cases addressed is given below in terms of tables and charts. The tests were carried out on a single processor 64 bit Windows 7 Home Edition system running an Intel® Pentium® CPU G640 @ 2.8 GHz, with 4.00 GB RAM. No networking was required for the testing.

It is to be noted that the maximum time needed for the tool to complete a scan and analysis of the entire application when all 24 malicious conditions were imposed on the tool took about 465 seconds to complete. The time was measured by echoing a timestamp

at entry to main() routine in the CAIS tool and another timestamp entry at the time of exit from main().

The tests were run multiple times to confirm that the time required to complete identical tests 2 or more times did not vary. The actual time for each run is used as data.

Another test run was also conducted using just 2 packages. The quantitative results are shown in Table 21 - Testing with 2 packages, Table 22 – Testing results with 3 packages, Table 23 – Testing results with 4 packages of an application, Table 24 – Testing results using one package containing Database Wrapper Classes and varying the number of secure tables from 1 to 10.

Table 21 Performance with 2 packages and varying the # of secure variables

# of Secure Variables	# of packages	# of Total Methods	# of CR Methods discovered	# of Source Code Files	# of Lines of Java Code	Tool Execution time (Seconds)
1	2	108	20	34	973	52
2	2	108	20	34	973	52
3	2	108	21	34	973	43
4	2	108	34	34	973	52
5	2	108	35	34	973	54
6	2	108	35	34	973	54
7	2	108	36	34	973	55
8	2	108	36	34	973	56
9	2	108	42	34	973	60
10	2	108	42	34	973	60
11	2	108	45	34	973	62
12	2	108	46	34	973	64
13	2	108	53	34	973	73
14	2	108	53	34	973	72
15	2	108	53	34	973	72
16	2	108	53	34	973	73
17	2	108	53	34	973	73
18	2	108	57	34	973	75
19	2	108	67	34	973	85
20	2	108	67	34	973	83
21	2	108	67	34	973	85
22	2	108	67	34	973	86
23	2	108	67	34	973	85
24	2	108	69	34	973	88
25	2	108	69	34	973	87
26	2	108	69	34	973	87
27	2	108	72	34	973	88

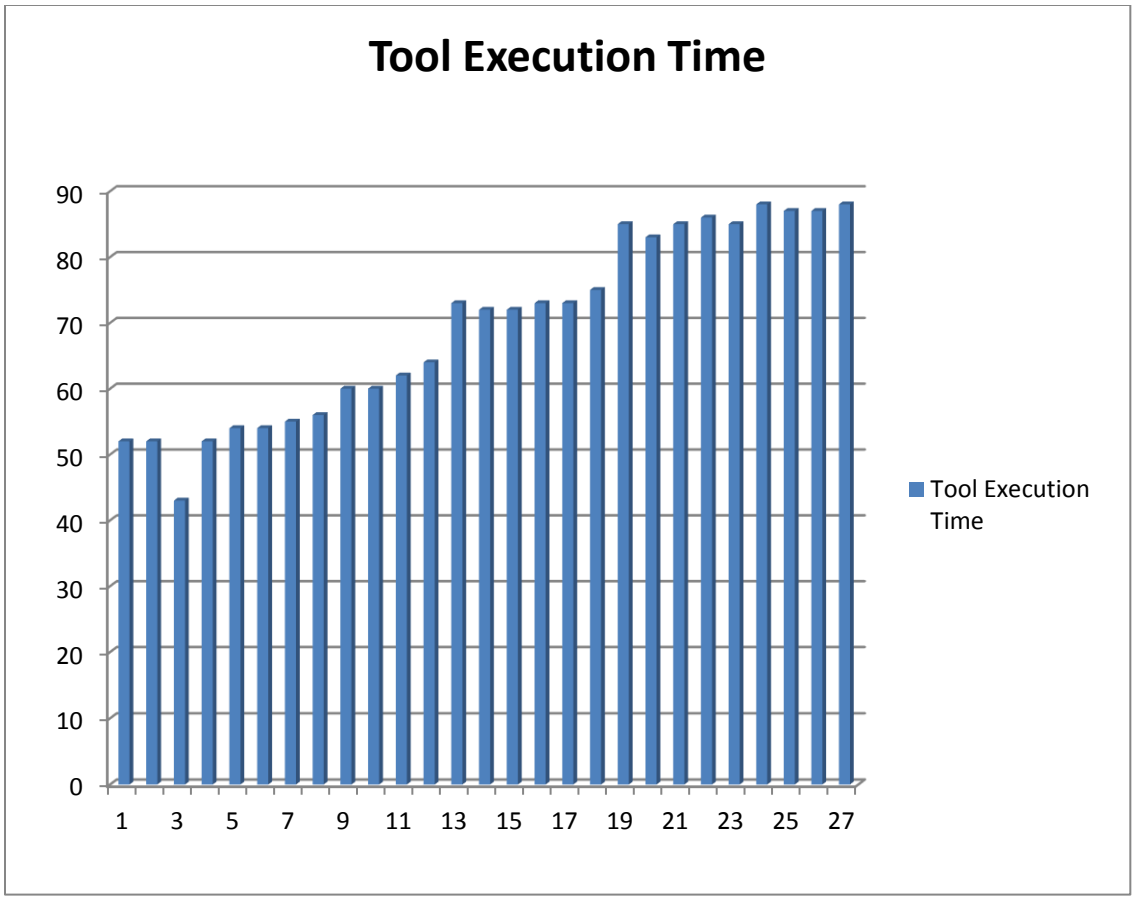


Figure 8 # of Secure Variables VS Tool Run Time for 2 pkgs (Table 21)

Analysis: The data from Table 21 lists the tool performance when the number of secure variables is varied from 1 to the maximum possible of 27. There is a proportionate increase in processing time as the number of secure variables are increased.

This relationship is depicted in the Figure 8.

Table 22 Performance with 3 packages and varying the # of secure variables

# of Secure Variables	# of packages	# of Total Methods	# of CR Methods discovered	# of Source Code Files	# of Lines of Java Code	Tool Execution time (Seconds)
1	3	115	20	17	1125	54
2	3	115	20	17	1125	55
3	3	115	21	17	1125	45
4	3	115	34	17	1125	55
5	3	115	35	17	1125	56
6	3	115	35	17	1125	57
7	3	115	36	17	1125	57
8	3	115	36	17	1125	59
9	3	115	42	17	1125	64
10	3	115	42	17	1125	62
11	3	115	45	17	1125	66
12	3	115	46	17	1125	68
13	3	115	53	17	1125	74
14	3	115	53	17	1125	75
15	3	115	53	17	1125	74
16	3	115	53	17	1125	75
17	3	115	53	17	1125	75
18	3	115	57	17	1125	78
19	3	115	67	17	1125	86
20	3	115	67	17	1125	88
21	3	115	67	17	1125	87
22	3	115	67	17	1125	89
23	3	115	67	17	1125	89
24	3	115	69	17	1125	89
25	3	115	69	17	1125	90
26	3	115	69	17	1125	89
27	3	115	69	17	1125	89
28	3	115	72	17	1125	92

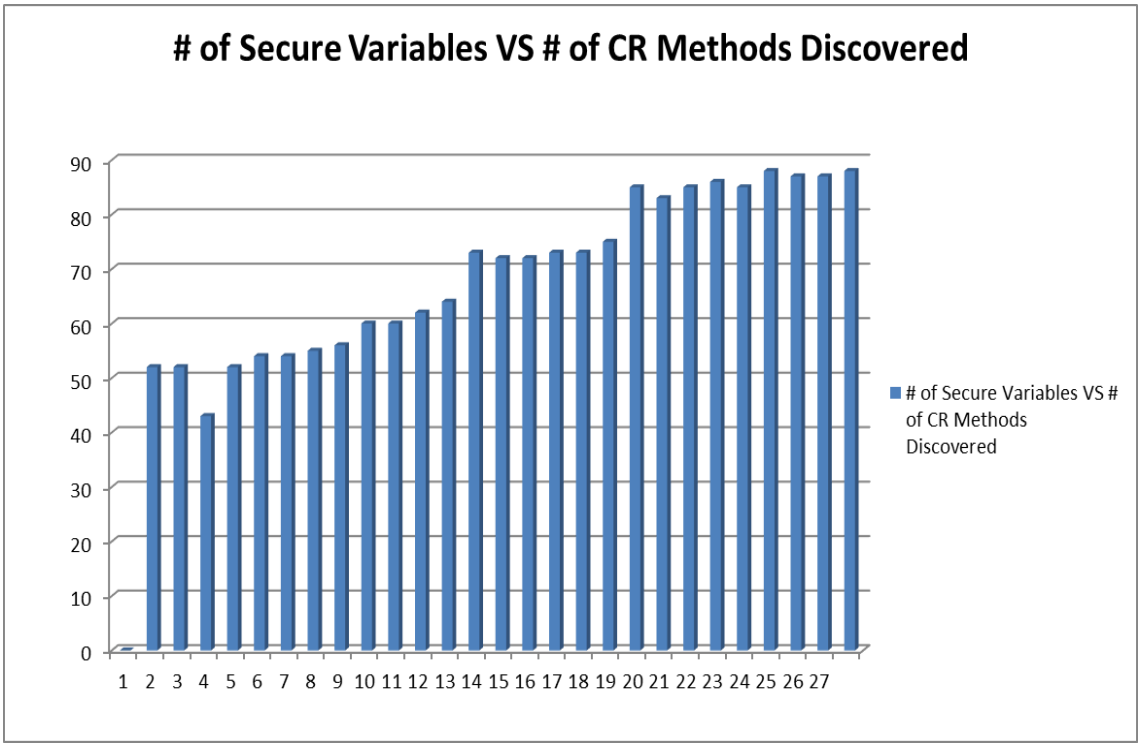


Figure 9 # of Secure Variables VS # of CR methods (3 pkgs) (Table 22)

Analysis A: Table 22 / Figure 9

The data in Table 22 reflects a linear relationship in processing time with 3 packages in the application. The tool processing time varies linearly when the number of Secure variables is increased from 1 to 28. Figure 9 depicts a sub relationship between the # of Secure Variables VS the # of CR methods discovered. This relationship is dependent on the nature of data.

Table 23 Performance with 4 pkgs and varying the # of secure variables/tables

# of Secure Variables/ Tables	# of packages	# of Total Methods	# of CR Methods discovered	# of Source Code Files/ Classes	# of Lines of Java Code	Tool Execution time (Seconds)
1	4	236	30	34	2870	177
2	4	236	30	34	2870	173
3	4	236	31	34	2870	192
4	4	236	42	34	2870	218
5	4	236	43	34	2870	225
6	4	236	43	34	2870	218
7	4	236	46	34	2870	227
8	4	236	49	34	2870	235
9	4	236	49	34	2870	240
10	4	236	55	34	2870	256
11	4	236	55	34	2870	256
12	4	236	59	34	2870	267
13	4	236	60	34	2870	271
14	4	236	67	34	2870	292
15	4	236	67	34	2870	295
16	4	236	67	34	2870	297
17	4	236	67	34	2870	295
18	4	236	68	34	2870	300
19	4	236	75	34	2870	314
20	4	236	89	34	2870	366
21	4	236	89	34	2870	365
22	4	236	89	34	2870	367
23	4	236	89	34	2870	373
24	4	236	89	34	2870	366
25	4	236	92	34	2870	372
26	4	236	92	34	2870	371
27	4	236	92	34	2870	372
28	4	236	94	34	2870	394
29	4	236	97	34	2870	382
30	4	236	100	34	2870	399
31	4	236	103	34	2870	398
32	4	236	106	34	2870	410
33	4	236	109	34	2870	420
34	4	236	112	34	2870	414
35	4	236	115	34	2870	416

Table 23 Cont'd						
# of Secure Variables/ Tables	**# of packages**	**# of Total Methods**	**# of CR Methods discovered**	**# of Source Code Files/ Classes**	**# of Lines of Java Code**	**Tool Execution time (Seconds)**
36	4	236	118	34	2870	434
37	4	236	121	34	2870	443
38	4	236	124	34	2870	440
35	4	236	115	34	2870	416
36	4	236	118	34	2870	434
39	4	236	134	34	2870	457
40	4	236	134	34	2870	465
41	4	236	135	34	2870	327
42	4	236	150	34	2870	355
43	4	236	151	34	2870	383
44	4	236	151	34	2870	370
45	4	236	152	34	2870	349
46	4	236	152	34	2870	357
47	4	236	152	34	2870	358
48	4	236	158	34	2870	370
49	4	236	158	34	2870	372
50	4	236	161	34	2870	390
51	4	236	162	34	2870	380
52	4	236	169	34	2870	418
53	4	236	169	34	2870	398
54	4	236	169	34	2870	418
55	4	236	169	34	2870	411

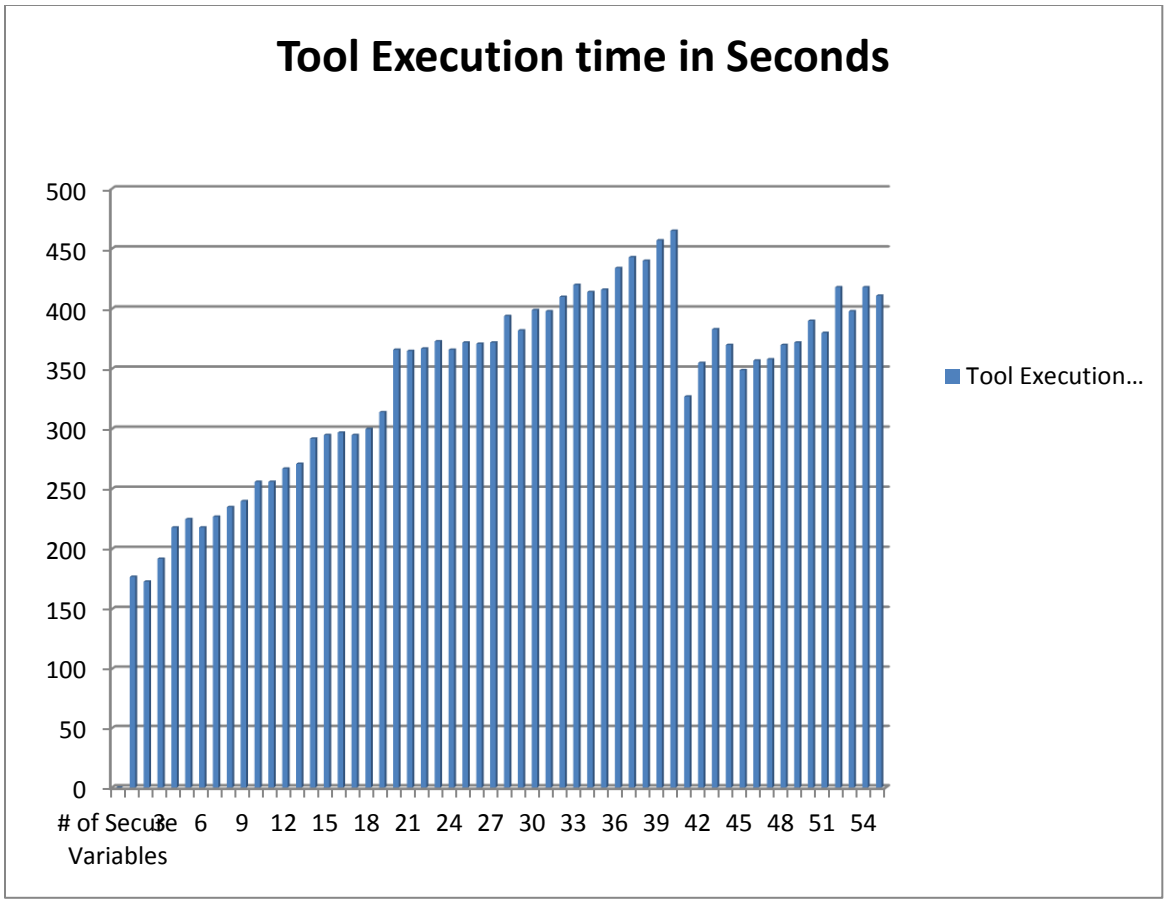


Figure 10 # of Secure Variables/Tables VS Tool Run Time for 4 pkgs (Table 23)

Analysis B: Table 23 / Figure 10

The data in Table 23 reflects the relationship between the # of secure variables and the processing time with 4 packages in the application. The tool processing time varies linearly when the number of Secure variables is increased from 1 to 55.

There is an aberration noted when the # of Secure variables is changed from 40 to 41. There is a drop in the processing time. The explanation for this is based in the nature of the proceesing in the analyzer section of the tool.

The analyzer processes all rules in a sequence. This sequence is contained inside a loop. The loop is repeated if any more CRs are discovered in a processing sequence. This is to ensure that no new CRs can be discovered in the next iteration.

When between 1 and 40 Secure Variables are listed in the SV.txt file, the Analyzer goes through 3 repetitions of the sequence before all the CR methods are discovered. The addition of the 41st secure variable result in the analyzer identifying the CR methods within 2 repetitions of the processing sequence. This explains why the time of execution for the tool drops at that juncture.

This reveals an interesting fact about the tool execution time, in that, the time required for processing the application depends very much on the nature of the source code in the application.

Figure 9 depicts the relationship between the # of Secure Variables VS the # of CR methods discovered. This relationship is also dependent on the nature of source code in the application.

Analysis C: Table 24 / Figure 11

The data in Table 24 reflects the relationship between the # of secure tables and the processing time with 1 package in the application that contains all of the Database Wrapper Classes (DBWC). There are 11 DBW Classes in the package out of which 10 are accessing a different secure table each. The tool processing time varies linearly when the number of Secure tables is increased from 1 to 10.

Figure 11 depicts the relationship between the # of secure tables and the execution time for the tool in seconds. It is an expectedly linear relationship.

Table 24 # of secure tables VS Tool execution time of 1 DBWC package

# of Secure Tables in DBWC	# of packages	# of Total Methods	# of CR Methods discovered	# of Source Code Files/ Classes	# of Lines of Java Code	Tool Execution time (Seconds)
1	1	50	3	11	1259	8
2	1	50	6	11	1259	9
3	1	50	9	11	1259	9
4	1	50	12	11	1259	9
5	1	50	15	11	1259	11
6	1	50	18	11	1259	11
7	1	50	21	11	1259	11
8	1	50	24	11	1259	11
9	1	50	27	11	1259	12
10	1	50	30	11	1259	13

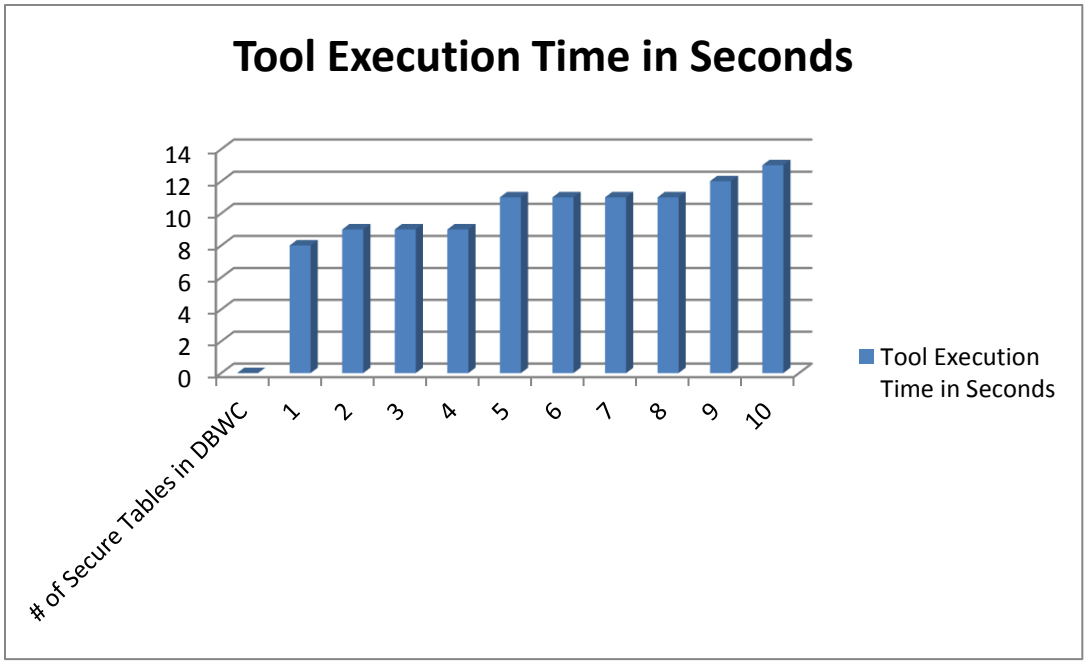
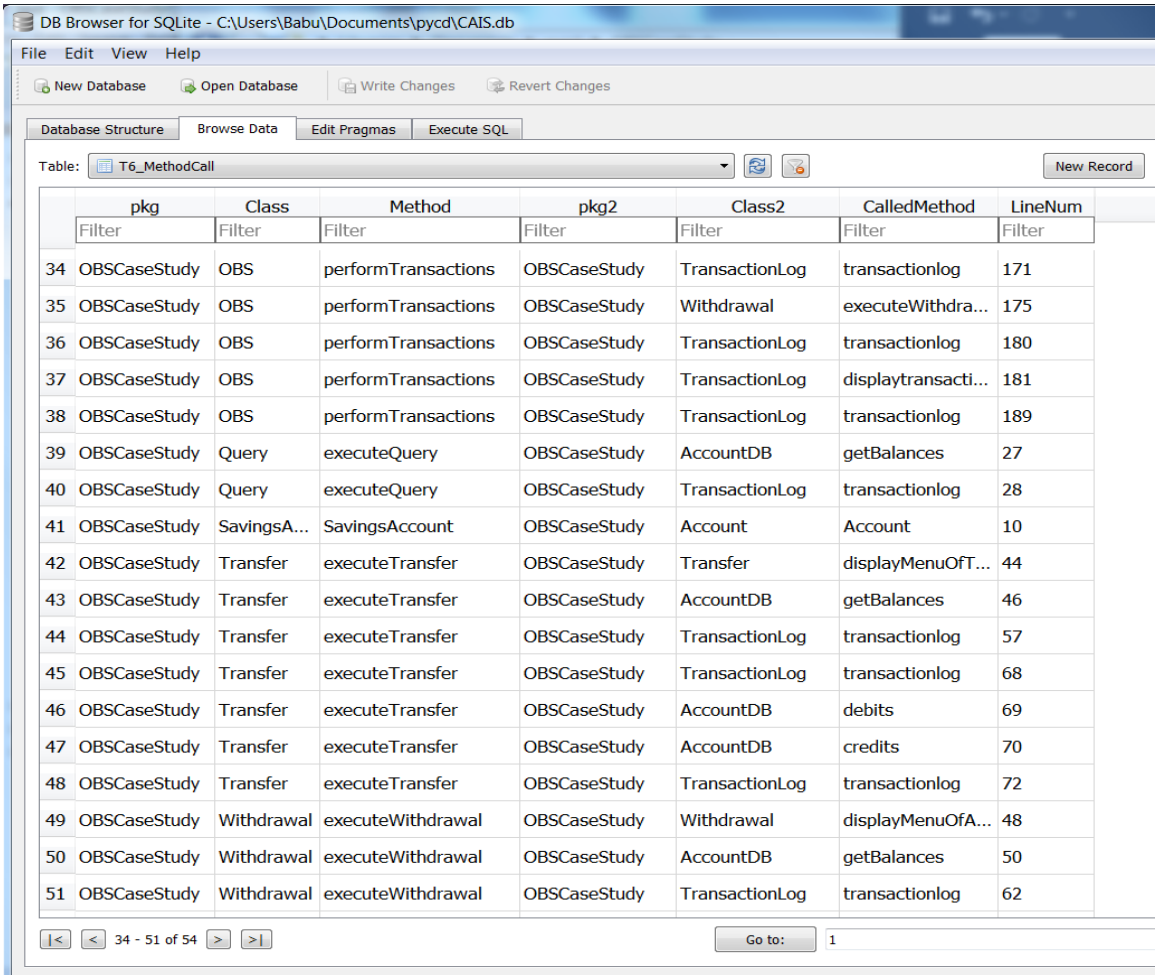


Figure 11 # of secure tables VS Tool Run time using one pkg (Table 24)

6.8. Observing Tool Database contents

The database contents build by the tool can be monitored by using a view into the tables that the tool builds. This is useful for understanding the entries that have been captured by the tool. For the purpose of monitoring the database, we should download an open source utility "DB Browser for SQLite" from the following link http://sqlitebrowser.org/ This utility allows us to open the CAIS database that is created after a run of the CAIS tool. It is possible to observe the contents of each table. A typical view from the DB Browser is shown below. This view shows the contents of Table T6_MethodCall which lists all of the methods of the application that call other methods in the application.



	pkg	Class	Method	pkg2	Class2	CalledMethod	LineNum
	Filter	Filter	Filter	Filter	Filter	Filter	Filter
34	OBSCaseStudy	OBS	performTransactions	OBSCaseStudy	TransactionLog	transactionlog	171
35	OBSCaseStudy	OBS	performTransactions	OBSCaseStudy	Withdrawal	executeWithdra...	175
36	OBSCaseStudy	OBS	performTransactions	OBSCaseStudy	TransactionLog	transactionlog	180
37	OBSCaseStudy	OBS	performTransactions	OBSCaseStudy	TransactionLog	displaytransacti...	181
38	OBSCaseStudy	OBS	performTransactions	OBSCaseStudy	TransactionLog	transactionlog	189
39	OBSCaseStudy	Query	executeQuery	OBSCaseStudy	AccountDB	getBalances	27
40	OBSCaseStudy	Query	executeQuery	OBSCaseStudy	TransactionLog	transactionlog	28
41	OBSCaseStudy	SavingsA...	SavingsAccount	OBSCaseStudy	Account	Account	10
42	OBSCaseStudy	Transfer	executeTransfer	OBSCaseStudy	Transfer	displayMenuOfT...	44
43	OBSCaseStudy	Transfer	executeTransfer	OBSCaseStudy	AccountDB	getBalances	46
44	OBSCaseStudy	Transfer	executeTransfer	OBSCaseStudy	TransactionLog	transactionlog	57
45	OBSCaseStudy	Transfer	executeTransfer	OBSCaseStudy	TransactionLog	transactionlog	68
46	OBSCaseStudy	Transfer	executeTransfer	OBSCaseStudy	AccountDB	debits	69
47	OBSCaseStudy	Transfer	executeTransfer	OBSCaseStudy	AccountDB	credits	70
48	OBSCaseStudy	Transfer	executeTransfer	OBSCaseStudy	TransactionLog	transactionlog	72
49	OBSCaseStudy	Withdrawal	executeWithdrawal	OBSCaseStudy	Withdrawal	displayMenuOfA...	48
50	OBSCaseStudy	Withdrawal	executeWithdrawal	OBSCaseStudy	AccountDB	getBalances	50
51	OBSCaseStudy	Withdrawal	executeWithdrawal	OBSCaseStudy	TransactionLog	transactionlog	62

Figure 12 SQLite Browser

CHAPTER 7

7. ONLINE BANKING CASE STUDY

7.1. The OBS Application Functional View

The OBS application case study is adapted from the book "Java, How to program", 6th Edition by Dietel and Dietel. The case study is enhanced by adding Transfer, Customer, CustomerDB, AcccountDB, TransactionLogDB, CheckingAccount and SavingsAccount objects. The case study originally used internal data structures to create accounts information. As a part of this case study all of the internal data structures representing account information have been moved over to database built using the Derby 10.X database. The On-Line Banking Case Study (OBS) is a small secure application that is representative of a commercial application for an online banking system.

The entry point to the application is through the OBSClient.java code which creates an OBS object and calls the run function on the object. This application is a secure application that allows a customer to login to the system using the Customer ID and a login PIN.

```
Welcome to OBS Online Banking System!

Please enter your CustomerID: XXXXXX
Enter your PIN: YYYYYY
```

If ID and PIN match, the user is logged in. If they do not match, then user is asked to enter ID and pin again. Once a user is logged in the system, it asks the user to select from the choice of accounts, either checking or savings.

```
Please select account type: 1 for Checking, 2 for Savings
```

Once the user selects the account type, the application serves up a menu that allows the customer to do the following. It asks the user to enter a choice from the menu.

```
Main Menu:

1 - View Customer Info
2 - View my balance
3 - Transfer funds
4 - Deposit funds
5 - Withdraw funds
6 - Exit
Enter a choice:
```

Once the user enters a choice then the application will allow him to operate the OBS as he desires. He should make the correct choices at each prompt and finally exit from the application.

The application was thoroughly tested to ensure that all the customer functionality such as Query, Deposit, Withdrawal and Transfer worked as designed. Customer can work on both checking and savings accounts and transfer money between the checking and savings s. The transaction log from an entire session is captured and stored in the database table "LogTable" for later review in case any abnormality is encountered in the application.

7.2. Classes in the OBS application

The schematic of the OBS application is shown below.

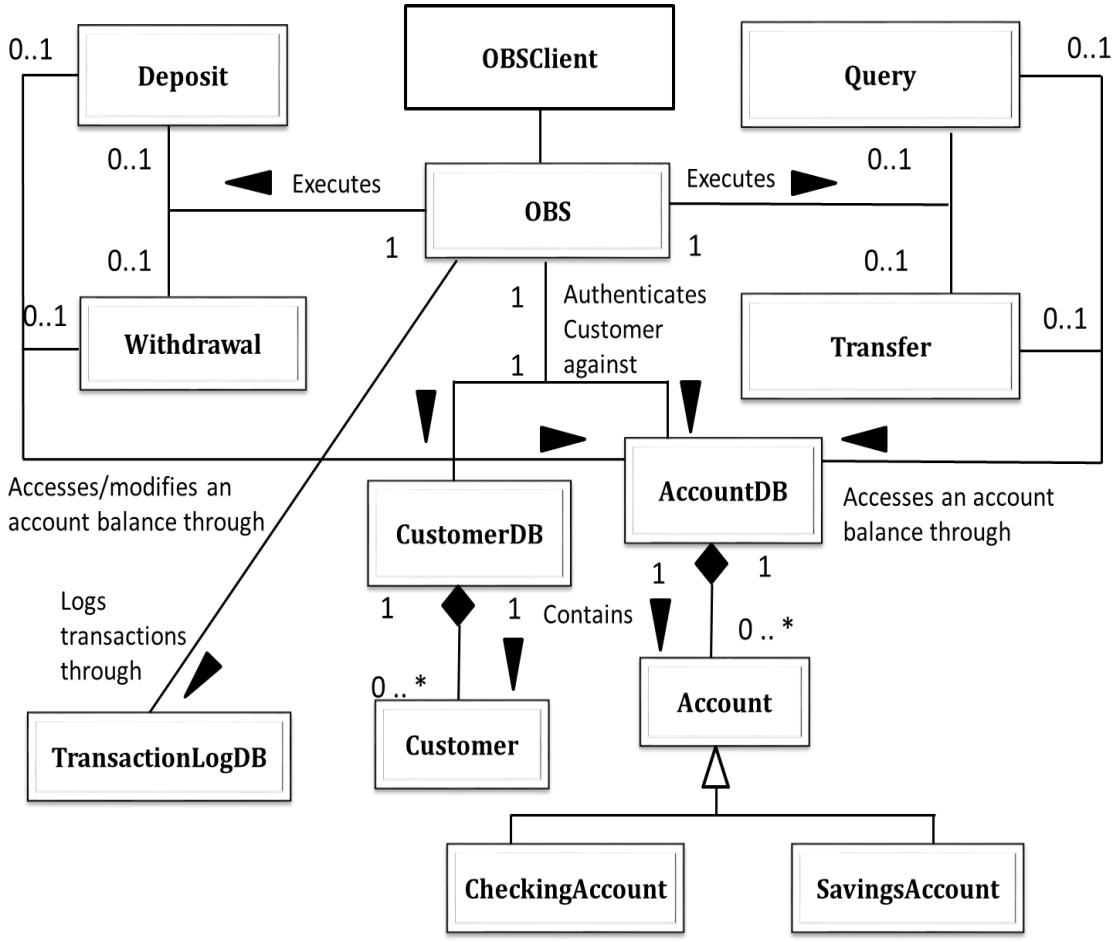


Figure 13 Class Diagram Schematic of the Online Banking System

The application consists of 13 different classes. The OBSClient is the class that is the entry point to the application. It calls OBS.run() method. The OBS class executes the Deposit, Withdrawal, Transfer and Query operations on the Customer Account. The customer account is created by the AccountDB class and is either a CheckingAccount or a SavingsAccount. The CustomerDB class creates the Customer. The CustomerDB class and the AccountDB class along with TransactionLogDB class are the Database wrapper classes which interact with the Derby 10.X database myDB that contains 4 tables. All

operations are logged by the transactionlog() operation. Each of the classes along with the attributes(variables) and the operations(methods) are listed. The case study source code is listed in Appendix B for easy reference.

1) OBSClient.java - 30 LOC

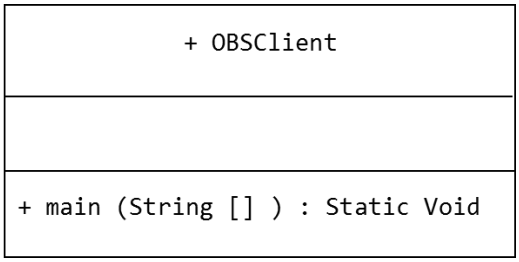


Figure 14 OBSClient class

OBSClient is the class that contains the main method. The main method creates an object of the OBS class and executes the OBS.run() operation

2) AccountDB.java - 162 LOC

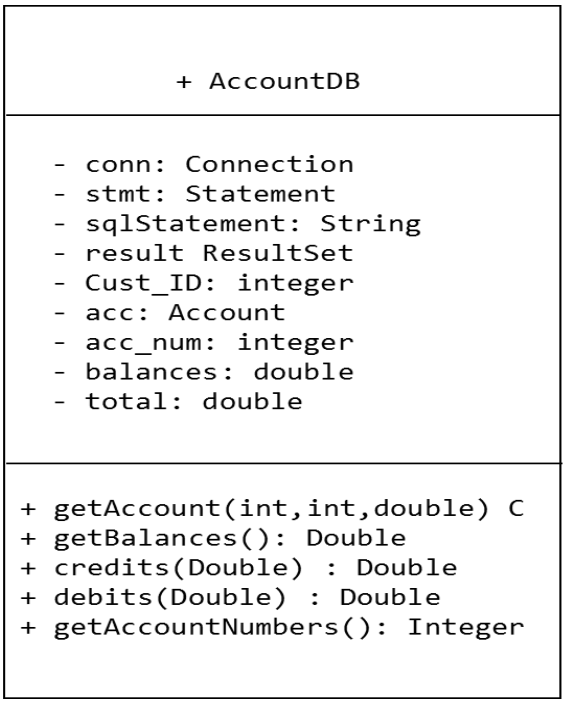


Figure 15 AccountDB Class

The AccountDB class is a Database Wrapper Class that holds account information from the SAVINGSTABLE or the CHECKINGTABLE. It creates objects of the Account Class with the information from the table. It has operations to getAccount(), getBalances, credits(), debits() and getAccountNumbers() from accounts.

3) OBS.java - 222 LOC

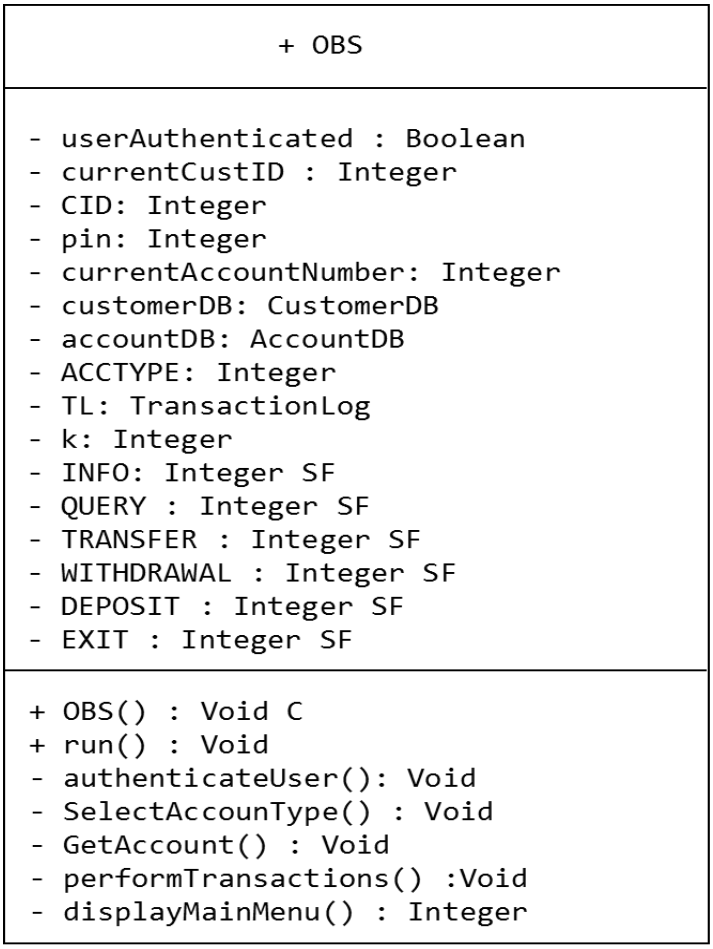


Figure 16 OBS class

OBS is the main coordinator class that gets user information from CustomerDB class and AccountDB classes, authenticates the user, selects the account type for the user to work

with, gets account information, displays Main OBS menu and performs the specified transactions.

4) Account.java - 57 LOC

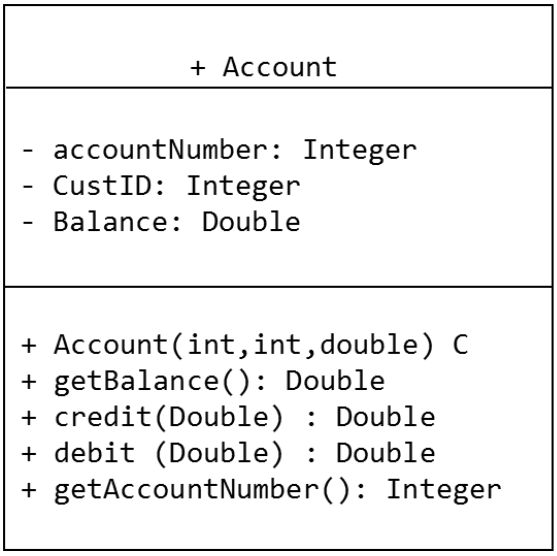


Figure 17 Account Class

The Account class is a parent class that holds account number, Customer ID and Balance. It declares a constructor to create the Account object, It has operations to getBalance(), credit(), debit() and getAccountNumber() on the Account class/object.

5) SavingsAccount.java - 16 LOC

The SavingsAccount class is an inherited class of parent Account, which holds savings account number, Customer ID and savings account balance through the creation and execution of a transaction on the account. It uses the super constructor to create the SavingsAccount object. It has operations to get balance, credit, debit from savings accounts and get account number.

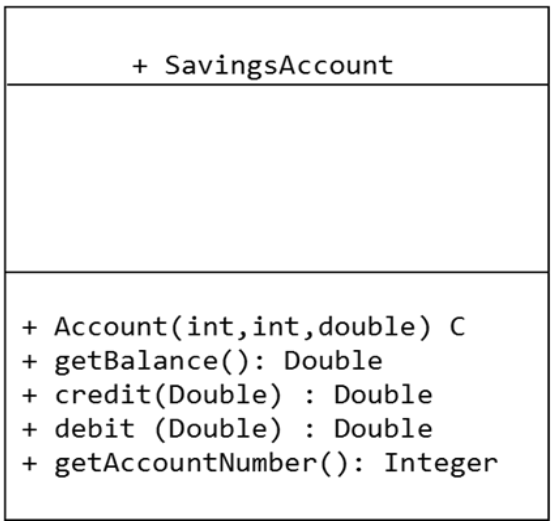


Figure 18 SavingsAccount Class

6) CheckingAccount.java - 18 LOC

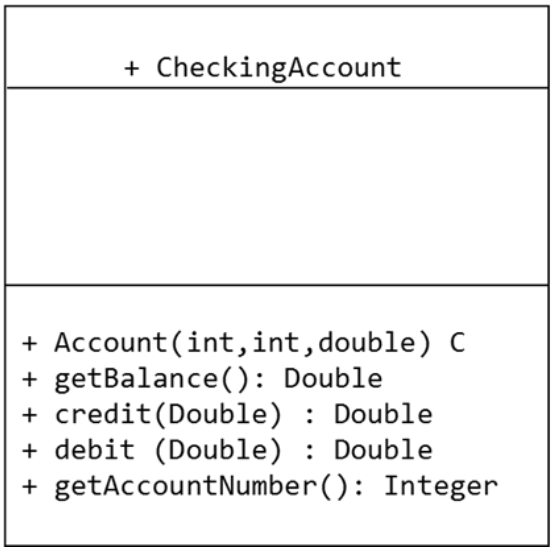


Figure 19 CheckingAccount Class

The CheckingAccount class is an inherited class of parent Account, which holds checking account number, Customer ID and checking account balance through the creation and execution of a transaction on the account. It uses the super constructor to create the CheckingAccount object. It has operations to get balance, credit, debit from checking accounts and get account number.

7) CustomerDB.java - 87 LOC

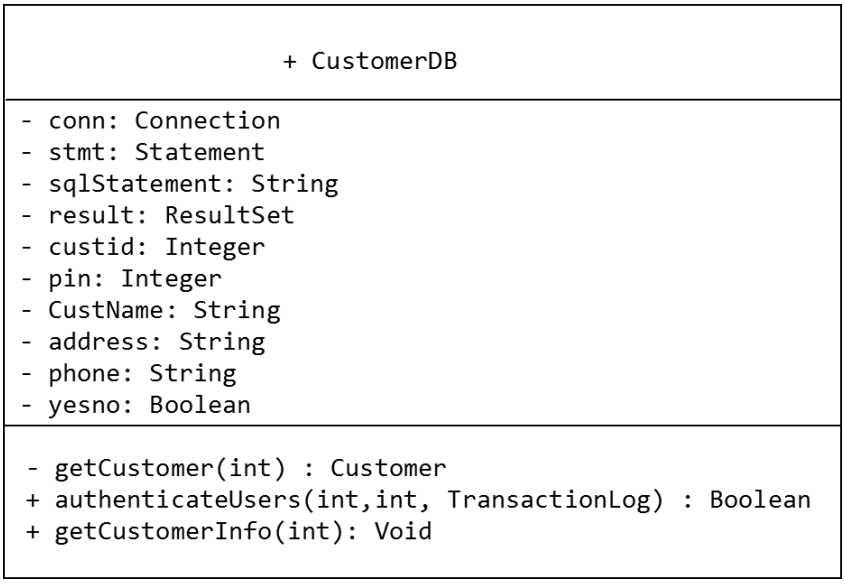


Figure 20 CustomerDB Class

The CustomerDB class is a database wrapper class that communicates with the database and with other objects in the application to retrieve data from the CUSTTABLE. It has well defined operations to access and manipulate data in specified tables. These operations are getCustomer(), authenticateUsers(), and getCustomerInfo()

8) Customer.java - 50 LOC

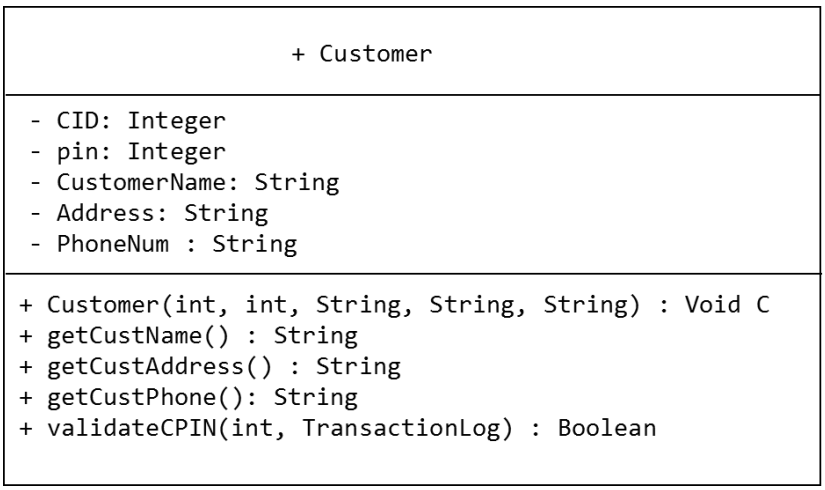


Figure 21 Customer Class

The Customer class is responsible for creating the object that holds Customer information during the time when a customer logs in to the system till such a time that the customer exits from the OBS. It retrieves customer information from the database tables through the database wrapper class CustomerDB. The class operations are getCustName(), getCustAddress(), getCustPhone(), validateCPIN()

9) Withdrawal.java - 157 LOC

The Withdrawal class is responsible for creating the Withdrawal object that holds account information during the time when a customer selects the withdrawal transaction in the system. The OBS object creates the Withdrawal object and calls the executewithdrawal() operation in response to the user selecting this operation from the OBS menu. The class operations are displayMenuOfAmounts(), and executeWithdrawal(). The executeWithdrawal operation is passed the TransactionLogDB object to record the transaction.

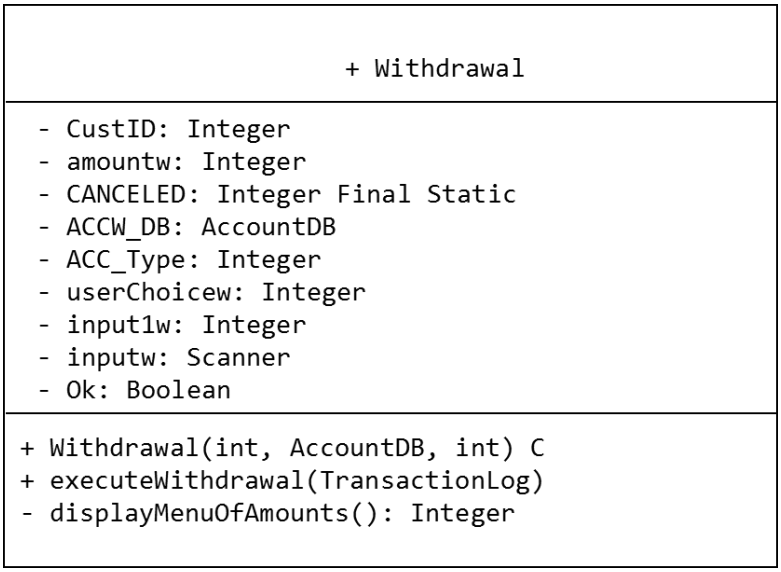


Figure 22 Withdrawal Class

10) Transfer.java - 149 LOC

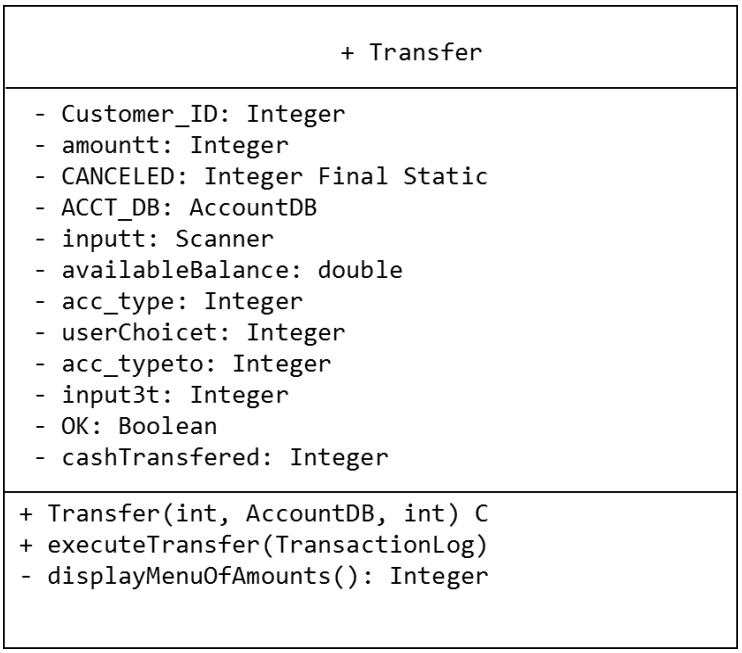


Figure 23 Transfer Class

The Transfer class is responsible for creating the Transfer object that holds account information during the time when a customer selects the transfer transaction in the system till such a time that the transaction is not completed. The OBS object creates the Transfer object and calls the executetransfer() operation in response to the user selecting this operation from the OBS menu. The class operations are displayMenuOfAmounts(), and executetransfer(). The executeTransfer operation is passed the TransactionLogDB object to record the transaction.

11) Deposit.java - 84 LOC

The Deposit class is responsible for creating the Deposit object that holds account information during the time when a customer selects the deposit transaction in the system till such a time that the transaction is not completed. The OBS object creates the Deposit

object and calls the executeDeposit() operation in response to the user selecting this operation from the OBS menu. The class operations are promptForDepositAmount(), and executeDeposit(). The executeDeposit operation is passed the TransactionLogDB object to record the transaction.

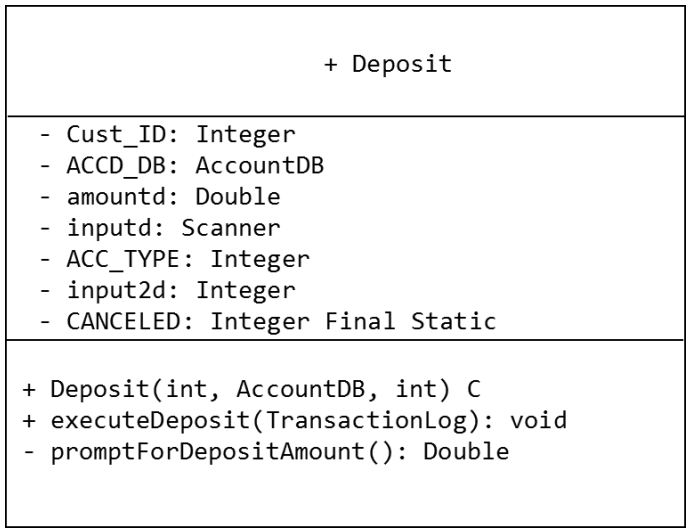


Figure 24 Deposit Class

12) Query.java - 38 LOC

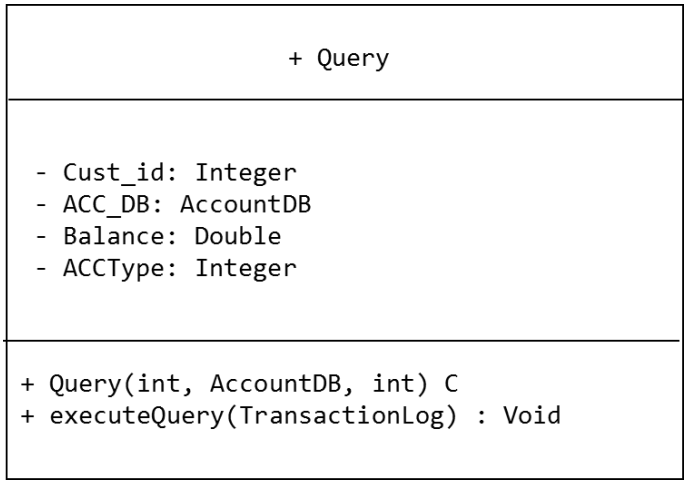


Figure 25 Query Class

The Query class is responsible for creating the Query object that holds account information during the time when a customer selects the deposit transaction in the system. The OBS object creates the Query object and calls the executeQuery() operation in response to the user selecting this operation from the OBS menu. The class operations are executeQuery(). The executeQuery operation is passed the TransactionLogDB object to record the transaction.

13) TransactionLogDB.java - 88 LOC

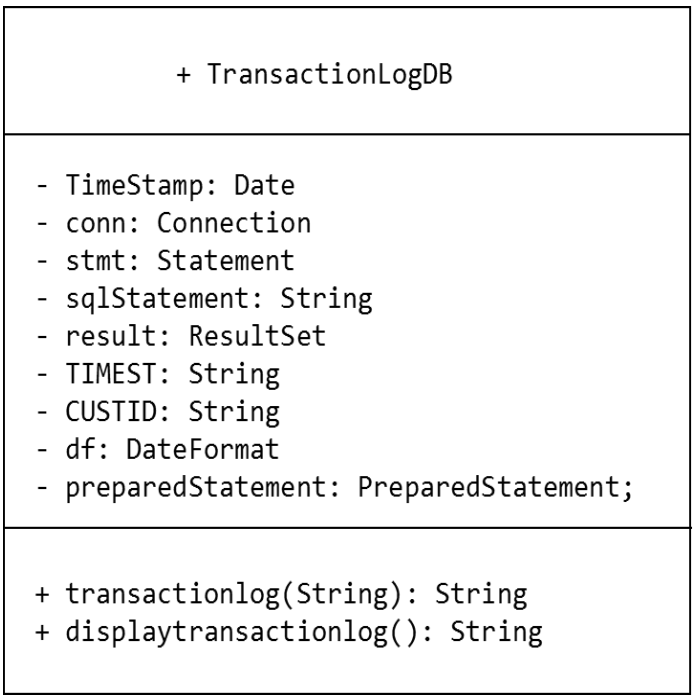


Figure 26 TransactionLogDB Class

The TransactionLogDB class is responsible for maintaining a transaction log for all of the transactions that take place when the OBS application is running. It stores information of the transactions in a transaction database table LOGTABLE. Each transaction is logged with its timestamp from the system along with the customer/user information and the operations selected from the menu by the user. The TransactionLogDB object is created

by OBS object and passed to the transaction operations. The class operations are transactionlog(), and displaytransactionlog() which can display the contents of the LOGTABLE.

The 13 classes in all contain 1158 lines of Java code including comment lines. A total of 40 methods are contained in the 13 classes.

Care was taken to ensure that simplistic java constructs were admissible as source code. These restrictions conform to the Coding standards mentioned in section 6.6.1 to ensure that the CAIS tool will be able to recognize and parse all the constructs in the application.

The database used in the application is Apache Derby Version 10.X. The database is named as "myDB2" and contains 4 separate tables, CUSTTABLE, CHECKINGTABLE, LOGTABLE and SAVINGSTABLE.

The CHECKINGTABLE contains the information about the Checking accounts for the Customer, the SAVINGSTABLE contain information about the savings account for the customer, the LOGTABLE records information about the transaction log and the CUSTTABLE contains customer information. The format of the tables is as given below and the secure data is identified by shading the column. LOGTABLE does not contain secure data.

Table 25 CUSTTABLE

CID	PIN	NAME	ADDRESS	PHONE
Integer	Integer	String 30 char	String 60 char	String 10 char

Table 26 CHECKINGTABLE

CUSTID	ACCNUM	BALANCE
Integer	Integer	Double

Table 27 SAVINGSTABLE

CUST_ID	ACCNUM	BALANCE
Integer	Integer	Double

Table 28 LOGTABLE

TIMEST	EVENTTYPE	CUSTID	LogStr
STRING 30 char	STRING 20 char	String 10 char	String 60 char

The tables are populated with 5 customers each with a PIN, NAME, ADDRESS and PHONE number. Each Customer is provided one checking and one savings account and a start balance of $ 1000.00 in the checking account and $ 2000.00 in the savings account.

The code was designed using the Eclipse IDE for java, compiled in Eclipse and the application was run under Eclipse.

7.3.Testing and Scope of Code Review.

Various test runs were conducted on the application. The testing included all the secure variables and tables in the list of secure variables recorded in the text file SV.txt.

Then a test run will determine the set of methods which are to be considered as CR methods to be reviewed. Subsequently malicious code can be added to the methods identified as Non CR methods. A repeat run of the tests should identify the additional methods which are now identified as CR methods.

It should be noted that within the methods designated as CR methods, the tool will provide detailed reasoning as to why the method is a CR method and list the security hot spots in that method.

The tool will also build a dynamic analysis that points out security hot spots to be analyzed at runtime. The results of the testing of the case study are mentioned below.

Test Run 1: With all secure variables and secure tables

1) SV.txt: Secure variables (35) and Secure Tables(3)

```
OBSCaseStudy Account CustID
OBSCaseStudy Account accountNumber
OBSCaseStudy Account Balance
OBSCaseStudy OBS ACCTYPE
OBSCaseStudy OBS k
OBSCaseStudy OBS currentCustID
OBSCaseStudy OBS CID
OBSCaseStudy OBS currentAccountNumber
OBSCaseStudy OBS pin
OBSCaseStudy Deposit Cust_ID
OBSCaseStudy Deposit amountd
OBSCaseStudy Deposit inputd
OBSCaseStudy Deposit input2d
OBSCaseStudy Withdrawal CustID
OBSCaseStudy Withdrawal amountw
OBSCaseStudy Withdrawal userChoicew
OBSCaseStudy Withdrawal input1w
OBSCaseStudy Transfer Customer_ID
OBSCaseStudy Transfer amountt
OBSCaseStudy Transfer input3t
OBSCaseStudy Transfer userChoicet
OBSCaseStudy AccountDB acc_num
OBSCaseStudy AccountDB balances
OBSCaseStudy CustomerDB pin
OBSCaseStudy CustomerDB CustName
OBSCaseStudy AccountDB total
OBSCaseStudy Customer pin
OBSCaseStudy Customer CID
OBSCaseStudy Customer CustomerName
OBSCaseStudy CheckingAccount CID
OBSCaseStudy SavingsAccount CID
OBSCaseStudy CheckingAccount AccNum
OBSCaseStudy SavingsAccount AccNum
OBSCaseStudy CheckingAccount ACCBalance
OBSCaseStudy SavingsAccount AccBalance
OBSCaseStudy CustomerDB CUSTTABLE
OBSCaseStudy AccountDB SAVINGSTABLE
OBSCaseStudy AccountDB CHECKINGTABLE
```

2) Time taken to scan and analyze the OBSCaseStudy code: 97 seconds
3) Total number of methods: 40
4) Total number of CR methods identified: 34
5) Total number of Dynamic (Runtime) spots identified: 28

Test Run 2: With all Secure Variables and Tables

Note: A Malicious piece of code added to a Non Secure method SavingsAccount() to tamper with the **three secure variables** as shown below:

```
SavingsAccount():

CID = 500005;
ACCNUM = 55555;
ACCBalance= 50000;
```

1) SV.txt: Same as in Test Run 1
2) Time taken to scan and analyze the OBSCaseStudy code: 92 seconds
3) Total number of methods: 40
4) Total number of Static CR methods identified: **35**
5) Total number of Dynamic (Runtime) spots identified: **29**

The SavingsAccount() constructor is identified as a CR method and the instructions that tampers with the secure data are flagged for review as shown below.

```
OBSCaseStudy-SavingsAccount-SavingsAccount Warning: 6-1A/--6-2A
Secure variable CID changed in Secure Class,
please review new CR method pkg OBSCaseStudy,class
SavingsAccount,method SavingsAccount at line number 11
```

```
OBSCaseStudy-SavingsAccount-SavingsAccount Warning: 6-1A/--6-2A
Secure variable ACCNum changed in Secure Class,
please review new CR method pkg OBSCaseStudy,class
SavingsAccount,method SavingsAccount at line number 12
```

```
OBSCaseStudy-SavingsAccount-SavingsAccount Warning: 6-1A/--6-2A
Secure variable ACCBalance changed in Secure Class,
please review new CR method pkg OBSCaseStudy,class
SavingsAccount,method SavingsAccount at line number 13
```

Test Run 3: With all Secure Variables and Tables

Note: A Malicious piece of code added to 2 Non Secure methods CheckingAccount() and SavingsAccount() to tamper with the **amounts** as shown below:

CheckingAccount():

```
CID = 500005;
ACCNUM = 55555;
ACCBalance= 50000;
```

SavingsAccount():

```
CID = 500005;
ACCNUM = 50505;
ACCBalance= 60000;
```

1) SV.txt: Same as in Test Run 1
2) Time taken to scan and analyze the OBSCaseStudy code: 89 seconds
3) Total number of methods: 40
4) Total number of Static CR methods identified: **36**
5) Total number of Dynamic (Runtime) spots identified: **30**

The CheckingAccount() method, and the SavingsAccount() constructor methods are identified as a CR method and the instructions that tamper with amount are flagged for review as shown below.

```
OBSCaseStudy-CheckingAccount-CheckingAccount Warning: 6-1A/--6-2A
Secure variable CID changed in Secure Class,
please review new CR method pkg OBSCaseStudy,class
CheckingAccount,method CheckingAccount at line number 11
```

```
OBSCaseStudy-CheckingAccount-CheckingAccount Warning: 6-1A/--6-2A
Secure variable ACCNum changed in Secure Class,
please review new CR method pkg OBSCaseStudy,class
CheckingAccount,method CheckingAccount at line number 12
```

```
OBSCaseStudy-CheckingAccount-CheckingAccount Warning: 6-1A/--6-2A
Secure variable ACCBalance changed in Secure Class,
please review new CR method pkg OBSCaseStudy,class
CheckingAccount,method CheckingAccount at line number 13
```

7.4.Analysis of the test results

The tests were carried out with all secure variables specified in the SV.txt file. The results can be interpreted as follows.

1) This is a small application and the nature of the application is such that the methods are highly coupled. Almost all classes have secure variables that they access or secure table data that is modified. As a result a high percentage of methods is a CR method. Any single method that is found to be a CR method

gives rise to a large number of CR methods because they are somehow related to the calling/called sequence.

2) For the few methods that were correctly identified as Non CR methods by the CAIS tool, a malicious code added to those methods immediately results in those methods being identified as CR methods.
3) The CAIS tool identifies all the security hotspots that must be manually reviewed. Each of these CR methods, the reason for them to be CR methods and each of the security hot spot at runtime are well documented by the tool giving the user easy to understand information that the reviewer can use for analysis.

Thus the CAIS tool does what it set out to do, namely, to provide a detailed picture of how a CR method is identified to be a CR method, how couplings result in additional CR methods being identified and how the run time security hot spots are identified. All of this information is valuable to analyzing the application from the perspective of a code review for security purposes.

CHAPTER 8

8. DISCUSSION

The proposed approach is different from taint analysis or string analysis, which detect the breach of the same integrity in a program. The static taint analysis checks any variable in a program that can be tampered by a user input, whereas dynamic taint analysis monitors the tainted input data from untrusted sources at runtime to track how the predefined tainted input propagates in a program. The string analysis detects the programs that use or propagate malicious user inputs without proper sanitization for the integrity of programs. However, our approach focuses on the malicious code in a program that might be added or changed by an insider attack, even though it aims at the same integrity of programs as taint analysis or string analysis.

The approach proposed in this dissertation has been developed by means of coupling, but it does not incorporate the IBCs for checking common coupling-based insider attacks. Common coupling might not be a popular programming style in object-oriented programming although it is allowed in the programming languages syntactically. This is because common coupling involves the use of global variables, which are not recommended for object-oriented programming. Instead of developing IBCs for common coupling, our approach assumes that common coupling is prohibited for secure software. Our approach might not cover all the security spots in a program against malicious insider attacks. The approach specifies the IBCs for identifying security spots in a program, but the IBCs defined in this dissertation might not be complete to detect all the possible insider attacks for compromising the integrity of a program. The programming skill is advancing and intelligent all the time. Accordingly, the insider attack means is

getting clever and crafty as well. Our approach may not be capable of detecting all the insider attacks.

Our approach has some limitations in terms of the target program that requires code review for verifying the integrity security. The approach in this dissertation focuses on sequential programs, but not on concurrent or parallel programs. Programs may be designed with multi-threads or multi-processes or they may run in parallel. Also, network related insider attacks are not analyzed to specify the IBCs against the attacks in our approach. These limitations can be addressed in future work.

One important aspect to consider is that damage from malicious attack happens at run time. The CAIS tool is to be run before compiling the application program. It ensures that malicious software is checked before it can be compiled into an executable. Thus, it acts as an early screening for integrity security purpose.

CHAPTER 9

9. CONCLUSION

This dissertation has described an approach that determines the scope of mandatory code review against the breach of integrity security committed by an insider attack. The approach points out the security spots in a program, each of which might be possibly exploited by a malicious insider to compromise the program. The approach specifies the IBCs using the concepts of coupling between objects in the object-oriented programming. In particular, the approach uses content, external, data/stamp, subclass, and package coupling to specify the IBCs.

Based on the data produced while testing the tool, we can conclude that the tool scans and analyzes a typical Java object-oriented secure application.

This dissertation contributes to mitigating the effort of code review by applying the code review to only the security spots of applications. The proposed approach would save the organization resources for developing secure applications and reduce the delivery time of products to the market. The approach provides a systematic means of detecting insecure code by drawing the reviewer's attention to the security spots, so it can help the reviewer who might have less expertise and experience in security field. In particular, this approach would be effective when the applications developed by outsourcing should be code-reviewed by the ordering bodies for validating the integrity security required for the applications.

This paragraph describes future work for determination of code review scope against insider attacks. The changeability/evolution of the CAIS tool is a primary concern for

designing the tool. Although we have exhaustively enumerated the integrity breach conditions that are criteria for identifying possible malicious codes in a program, new integrity breach conditions might need to be added to the tool later. Additionally, the capability of the tool could be extended to multithreaded applications, GUI based applications, Web based applications with more advanced Java features. The CAIS tool can also be extended to include other object oriented languages such as C++ and Python.

The IBCs might need to be revised in order to reduce the false positive and false negative errors. A Non CR method may be identified and included in the scope of the code review (false positive) while a CR method may not be identified as a Non CR method (false negative). These errors should be reduced.

REFERENCES

[Smith06] G. Smith, "Principles of Secure Information Flow Analysis", Malware Detection, Springer, Vol. 27, pp. 291-307, 2007

[Jovanovic06] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Applications Vulnerabilities (Short Paper)", IEEE Symposium on Security and Privacy, Washington DC, pp. 258-263, 2006.

[Newsome04] J. Newsome, and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software", School of Computer Science, Carnegie Mellon University, 2004.

[Yu14] F. Yu. M. Alkhalaf, T. Bultan, and O. H. Ibarra, "Automata-Based Symbolic String Analysis for Vulnerability Detection", Formal Methods in System Design, Vol. 44, Issue 1, February, 2014.

[Lee04] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking", Proceedings of the 11th ACM International Conference on Architectural Support for Programming Language and Operating Systems, Boston, Massachusetts, October 9-13, 2004, pp. 85-96.

[Yin07] H. Yin, D. Song, M. Egele, M. Kruegel, C. Kirda, E., "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis", 14th ACM Conference on Computer and Communication Security, Alexandria, Virginia, USA, October 29 - November 2, 2007, pp. 116-127.

[Warmer03] J. Warmer, and A. Kleppe, "The Object Constraint Language: Getting Your Models Ready for MDA", Addison-Wesley, 2nd Edition, 2003.

[Collins13] M. Collins, D. M. Cappelli, T. Caron, R. F. Trzeciak, and A. P. Moore, "Spotlight On: Programmers as Malicious Insiders-Updated and Revised", CERT, Software Engineering Institute, Carnegie Mellon, December 2013.

[Silowash12] G Silowash, D. Cappelli, A. Moore, R. Trzeciak, T. J. Shimeall, and L. Flynn, "Common Sense Guide to Mitigating Insider Threats, 4th Edition*"* CERT, Software Engineering Institute, Carnegie Mellon, December 2012.

[CERT13] CERT, http://www.cert.org/insider_threat/study.html, 2013.

[Pfleeger09] S. L. Pfleeger and J. M. Atlee, "Software Engineering: Theory and Practice", Fourth Edition, Prentice Hall, 2009.

[Shin10] M. E. Shin, N. Patel, and S. Sethia, "Detection of Malicious Software Engineer Intrusion," 22st International Conference on Software Engineering and Knowledge Engineering (SEKE'2010), San Francisco, July 1-3, 2010.

[Shin11] M. E. Shin, S. Sethia, and N. Patel, "Component-based Malicious Software Engineer Intrusion Detection," IEEE 5th International Conference on Secure Software Integration and Reliability Improvement (SSIRI), Jeju, Korea, June 27-29, 2011.

[Pattabiraman09] K. Pattabiraman, N. Nakka, Z. Kalbarczyk and R. Iyer, "Discovering Application-Level insider attacks using symbolic execution," University of Illinois at Urbana Champaign, January 2009.

[Gomaa2000] H. Gomaa, "Designing concurrent, distributed, and Real-Time applications with UML", Addison-Wesley, 2000.

Code Conventions for the JavaTM Programming Language
(http://java.sun.com/docs/codeconv/)

Java Programming Style Guidelines (http://geosoft.no/javastyle.html)

Java, How to Program, Dietel & Dietel 6th Edition

APPENDIX A - DESIGN OF THE CODE ANALYSIS TOOL

The following figures show the high-level schematic of the Python code base. The modules that are highlighted in Bold are modules that call other functions. The entire code is written up in the described modules with each module handling a certain function of the overall code scanning and analysis tool. The high-level description of the modules in provided in the next section. Each module is described in details and its primary tasks are listed out.

The Integrity Security Relation Database is a set of tables that are built by the CodeScanner as it scans the java code files.

The CodeAnalyzer uses the information in the table alongwith the Integrity breach conditions and the secure variable/table information provided by the architect to decide which methods would constitute CR (Code Review) methods.

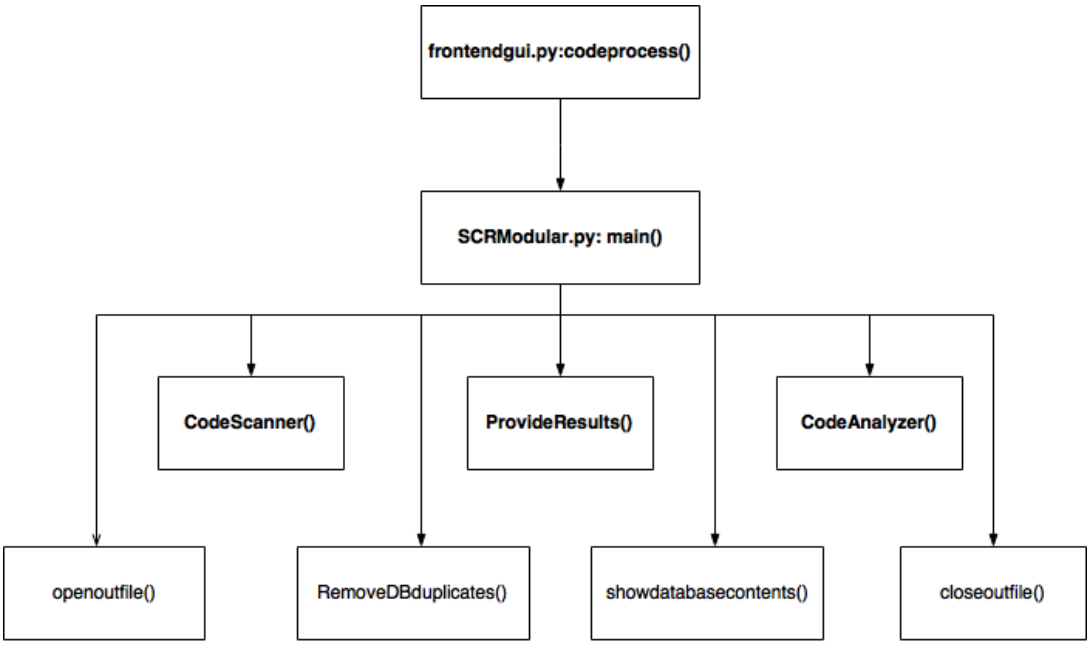


Figure 27 Top level module codeprocess()

i. frontendgui :codeprocess() function

Usage: frontendgui()

Input: None

Output: None

i. The frontendgui module sets the formatting details of the GUI window class.
ii. It opens the GUI for the user to enter the information that allows the scanner to start a scan
iii. User enters the full path location of the file containing the list of secure variables and secure table.
iv. The user enters the full path of the location of the Application packages
v. The user enters the list of packages separated by a space
vi. When the Start button is pressed, the module calls codeprocess()
vii. codeprocess() calls the main function in the SCRModular.py file passing the 3 variables as arguments .

ii. SCRModular.py main() function:

Usage: main(secfile,pkgloc,pkglist)

Input: SecureVariablelist file, packagelocation, listofpackages

Output: None

i. Main sets the logger file for the entire scan and analysis
ii. Main calls openoutfilebackup () to write scanning output
iii. Main calls the CodeScanner() passing 3 arguments from frontendgui()
iv. Main calls closeoutfilebackup() to close the file recording the scanning output
v. Main calls RemoveDBduplicates(0 to remove duplicates from a set of tables
vi. Main class showdatabasecontents to view the contents in tables
vii. Main class openoutfile() which is writes analysis information for the user to view

viii. Main calls CodeAnalyzer()
ix. Main calls RemoveDBduplicates() and showdatabasecontents() again
x. Main calls ProvideResults()
xi. Main calls RemoveDBduplicates once more()
xii. Main calls closeoutfile() to close the output file to the user.
xiii. Main returns()

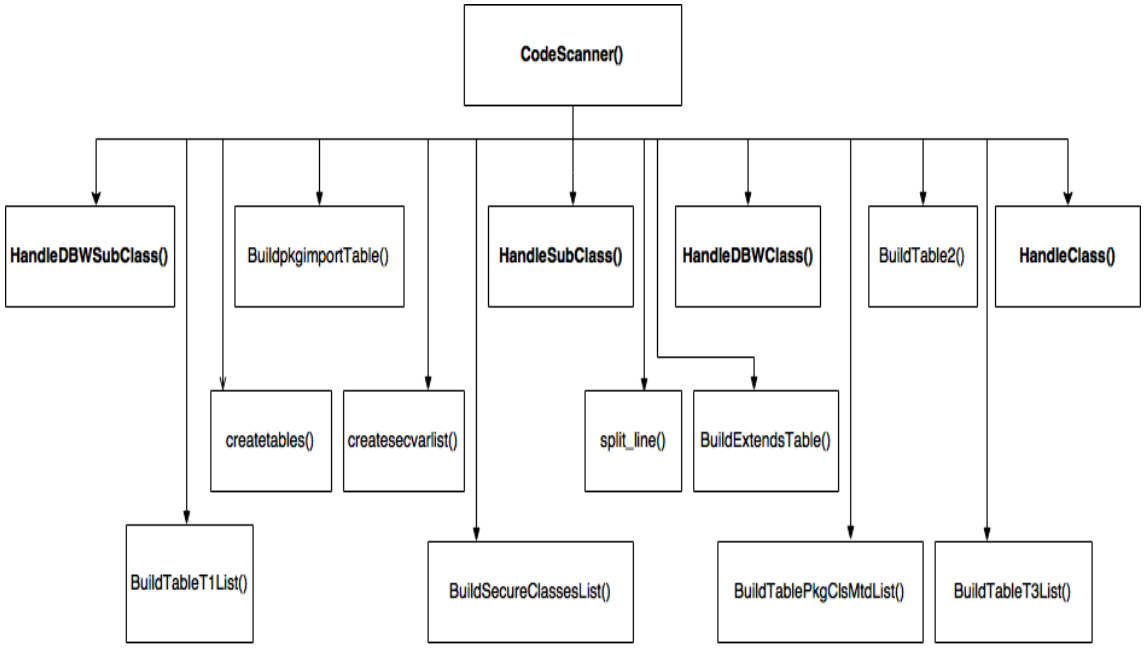


Figure 28 CodeScanner() module

iii. SCRModular.py CodeScanner() function:

Usage: CodeScanner(secfile,pkgloc,pkglist)

Input: SecureVariablelist file, packagelocation, listofpackages

Output: None

i. CodeScanner sets all the global variables needed for the scanning process
ii. CodeScanner sets up local list data structures to store certain cached database table contents

iii. CodeScanner sets up lists of ignorewords , keywords DBKW words which will be used and skipped as needed for certain code scanning.
iv. CodeScanner creates all of the database tables required to scan and analyze the code by calling createtables()
v. CodeScanner creates the secure variables list by calling createsecvarlist() and storing the secfile it received in Table T2_PkgCls using BuildTable2() function
vi. CodeScanner outputs basic scanning messages to the Outfile
vii. CodeScanner sets up the list of packages to scan based on pkgloc and pkglist
viii. CodeScanner extracts the names of the files in the packages in the given folder/directory
ix. **FOR Each** Package and **FOR Each** file in the package:

- CodeScanner starts the scan of the java source files from the first package. It will scan and process each line of the java source file, strip the line of blanks and splits the tokens into a list of tokens. After each line is scanned it will process the line based on the login given in the bulleted points and continue scanning till it reaches the end of file at which point it will start the scan of the next java source file in that package.

- CodeScanner looks for certain tokens and calls routines to process further based on the tokens read.
- If CodeScanner scans token “package” it will record the package being scanned
- If CodeScanner scans token “import” it will build pkgimport table accordingly
- If CodeScanner scans token “import” and “java” and “sql” it will set DBWC
- If CodeScanner scans tokens “public” and “class” and “classname” for the given source file, then it will Build Table T2_PkgCls
    - If DBWC is True then call HandleDBWClass() for further scanning
    - Else call HandleClass() for further scanning

- If CodeScanner scans token “extends” then it will add parent and child class entries in the Extends Table and call HandleSubClass() to handle further scanning
- CodeScanner scans the next line of source code, strips the line of blanks
- **END FOR**

x. CodeScanner builds the local lists for Table T1_SecureVar by calling BuildTableT1_SecureVarList()
xi. CodeScanner builds the local lists for Table T3_Var_Modifier by calling BuildTableT3_Var_ModifierList()
xii. CodeScanner builds the local lists for Table SecureClass by calling BuildTableSecureClassesList()
xiii. CodeScanner builds the local lists for Table PkgClsMtdList by calling BuildTablePkgClsMtdList()
xiv. CodeScanner writes debug messages and returns()

iv. CodeAnalyzer() Module:

Usage: CodeAnalyzer()

Input: None

Output: None

The code analyzer module starts analyzing once the CodeScanner() has scanned the entire code and stored the constructs in the various tables based on their syntax. The CodeAnalyzer() analyzes the tables and using its inbuilt intelligence algorithms applies these to the tables and builds the CRMethod table. The UCRMethod table is a simple version of the CRMethod table and contains unique CR methods.

i. CodeAnalyzer records entry into the routine
ii. CodeAnalyzer initializes 2 variables TotalCRBefore =0 and TotalCRAfter=1
iii. **WHILE** TotalCRAfter > TotalCRBefore **DO**

- CodeAnalyzer calls ProcessResultsTable6()
- CodeAnalyzer calls ReviewTable6contents()
- CodeAnalyzer calls ProcessTable4_6_1A()
- CodeAnalyzer calls ProcessTable5_1_6_1B()

- CodeAnalyzer calls ProcessTable6_1C()
- CodeAnalyzer calls ProcessTable6_6_1D()
- CodeAnalyzer calls ProcessTable5_20_6_1F()
- CodeAnalyzer calls ProcessTable8_1_6_1F()
- CodeAnalyzer calls ProcessTable7()
- CodeAnalyzer calls ProcessTable6_3A()
- CodeAnalyzer calls ProcessTable6_3B()
- CodeAnalyzer calls ProcessTable8_2_for_6_3C()
- CodeAnalyzer calls ProcessTable6_3D()
- CodeAnalyzer calls ProcessTable6_3E()
- CodeAnalyzer calls ProcessTable6_6_3E()
- CodeAnalyzer calls ProcessTable4_6_4A_B()
- CodeAnalyzer calls ProcessVariableRule6_4C_Use()
- CodeAnalyzer calls ProcessVariableRule6_4C_Change()
- CodeAnalyzer calls ProcessMethod6_4C()
- CodeAnalyzer calls ProcessTable6_6_5A()
- CodeAnalyzer calls ProcessTable6_6_5B()
- CodeAnalyzer calls ProcessResultsTable6()
- CodeAnalyzer calls RemoveDuplicatesCRMethod()
- CodeAnalyzer sets TotalCRAfter equal to the number of unique CR methods discovered during the iteration
- **END WHILE**

iv. CodeAnalyzer calls ProcessDynamicRule1()
v. CodeAnalyzer calls ProcessDynamicRule2()
vi. CodeAnalyzer calls ProcessDynamicRule3()
vii. CodeAnalyzer records exit time and returns()

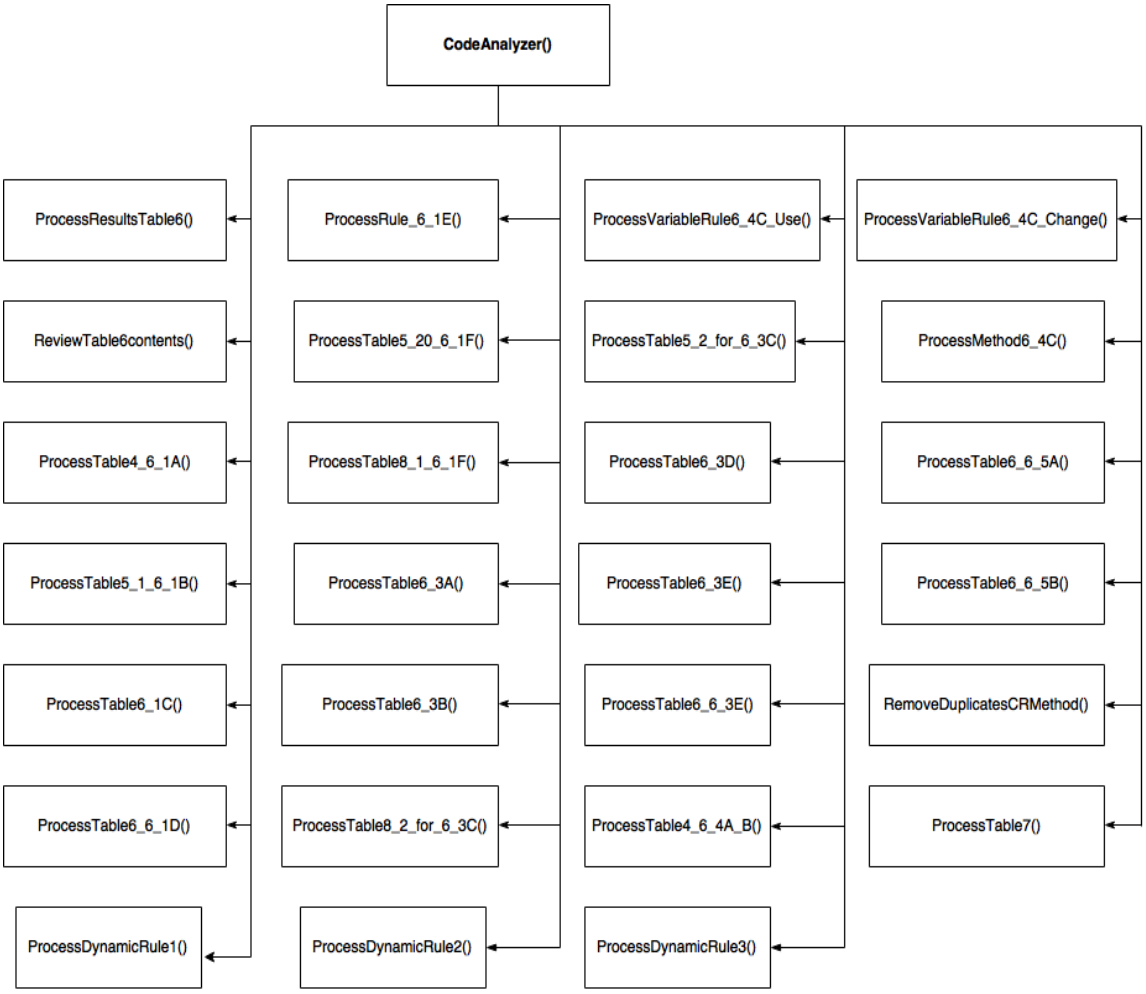


Figure 29 CodeAnalyzer() module

Note here that all of the processing of Tables by the code analyzer is described in details in the Section of Creation and processing algorithms for all the tables. Care is taken to indicate the number code mentioned in this section with the specific algorithm that is identified with the same number code. No other design details would be provided for these ProcessTable functions.

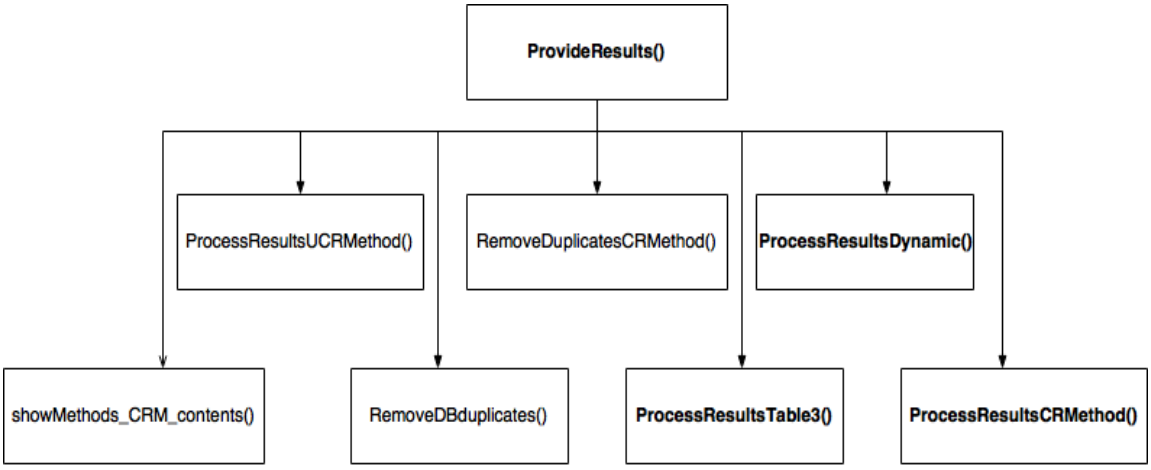


Figure 30 ProvideResults() module

v. ProvideResults() Module:

Usage: ProvideResults()

Input: None

Output: None

This module only works to provide the user with the information that is stored in the CRMethod, UCRMethod tables, Table T3_Var_Modifier and the Dynamic Message table. It is the final step for displaying information in a formatted manner. It calls a set of routines to do the same

```
 i.   ProvideResults calls ProcessResultsTable6()
 ii.  ProvideResults calls RemoveDBduplicates()
 iii. ProvideResults calls ProcessResultsTable6()
 iv.  ProvideResults calls RemoveDBduplicates()
 v.   ProvideResults calls showMethods_CRM_contents()
 vi.  ProvideResults calls ProcessResultsTable3()
 vii. ProvideResults calls ProcessResultsCRMethod()
 viii. ProvideResults calls ProcessResultsDynamic()
 ix.  ProvideResults returns
```

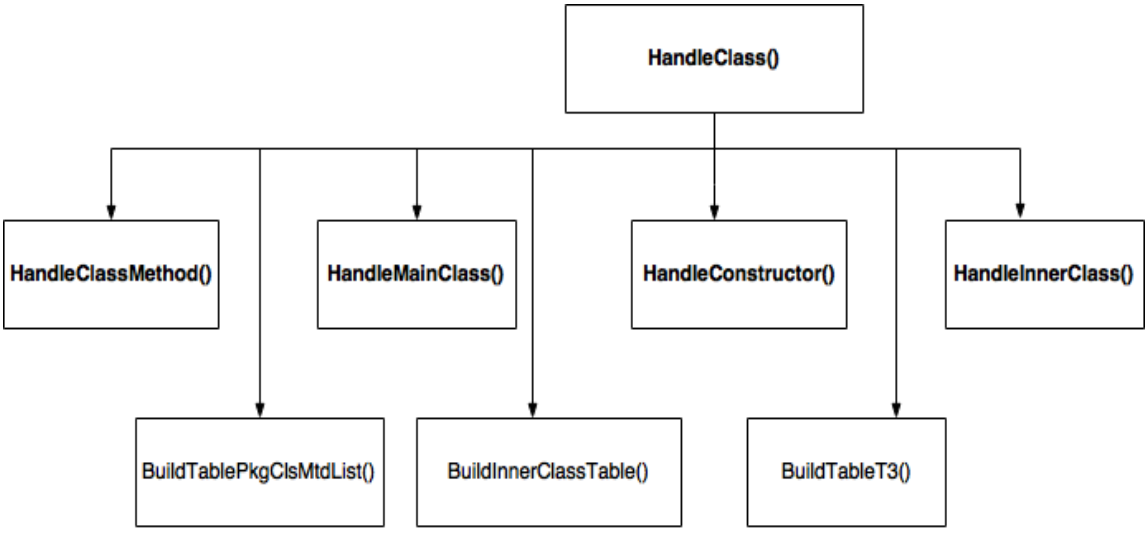


Figure 31 HandleClass() module

vi. HandleClass Module:

Usage: HandleClass(classname, srccode)

Input: Classname, sourcefile

Output: None

This module is handed control by CodeScanner() when it detects that a class declaration has been encountered. It is passed 2 arguments classname, which is the name of the class to be processed and srccode, which is the java source file name being processed.

i. The HandleClass() sets global and local variables for processing a Class.
ii. The HandleClass module scans the entire Class code in that source file. It scans line by line to handle the local variable declarations and also parses the constructor/inner class and methods declared within that Class.
iii. It skips any line that is a blank or starts with a comment //. While skipping or otherwise, it always increments the global linecount by 1 when moving to the next line.
iv. When it encounters a variable declarations, it calls BuildTableT3_Var_Modifier() to store those in Table T3_Var_Modifier

v. When it encounters a constructor (a special method) it calls BuildPkgClsMtdTable() to store the tuple Pkg, Class, Method in the PkgClsMtd table and calls the HandleConstructor() module to handle further processing of a constructor declaration.

vi. When it encounters an inner class declaration, it calls the BuildInnerClassTable() to build the InnerClass Table and HandleInnerClass() for further processing of the inner class of that class.

vii. When it encounters a method declaration, it calls BuildPkgClsMtdTable() to store the tuple Pkg, Class,Method in the PkgClsMtd table and HandleClassMethod() module for further processing of the method.

viii. When it encounters a main class declaration, it calls the HandleMainClass() module for further processing of the main routine (dynamic analysis) in that class.

ix. Once execution is complete the module hands control back to the CodeScanner() module.

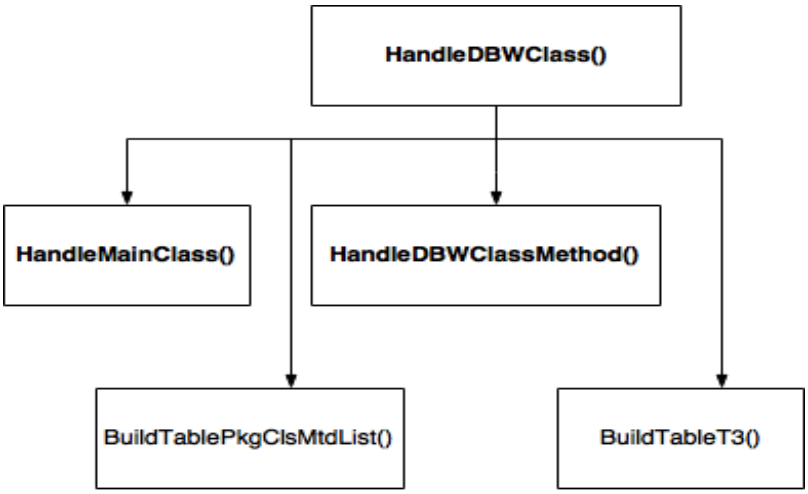


Figure 32 HandleDBWClass() module

vii. HandleDBWClass Module:

Usage: HandleDBWClass(DBWclassname, srccode)

Input: DBWClassname, sourcefile

Output: None

This module is handed control by CodeScanner() when it detects that a class declaration has been encountered. It is passed 2 arguments classname, which is the name of the class to be processed and srccode, which is the java source file name being processed.

i. The HandleDBWClass() sets global and local variables for processing a DBW Class.
ii. The HandleDBWClass module scans the entire DBW Class code in that source file. It scans line by line to handle the local variable declarations and also parses the constructor/inner class and methods declared within that Class.
iii. It skips any line that is a blank or starts with a comment //. While skipping or otherwise, it always increments the global linecount by 1 when moving to the next line.
iv. When it encounters a variable declaration, it calls BuildTableT3_Var_Modifier() to store those in Table T3_Var_Modifier
v. When it encounters a method declaration, it calls BuildPkgClsMtdTable() to store the tuple Pkg, Class,Method in the PkgClsMtd table and HandleDBWClassMethod() module for further processing of the method.
vi. When it encounters a main class declaration, it calls the HandleMainClass() module for further processing of the main routine (dynamic analysis) in that class.
vii. Once execution is complete the module hands control back to the CodeScanner() module.

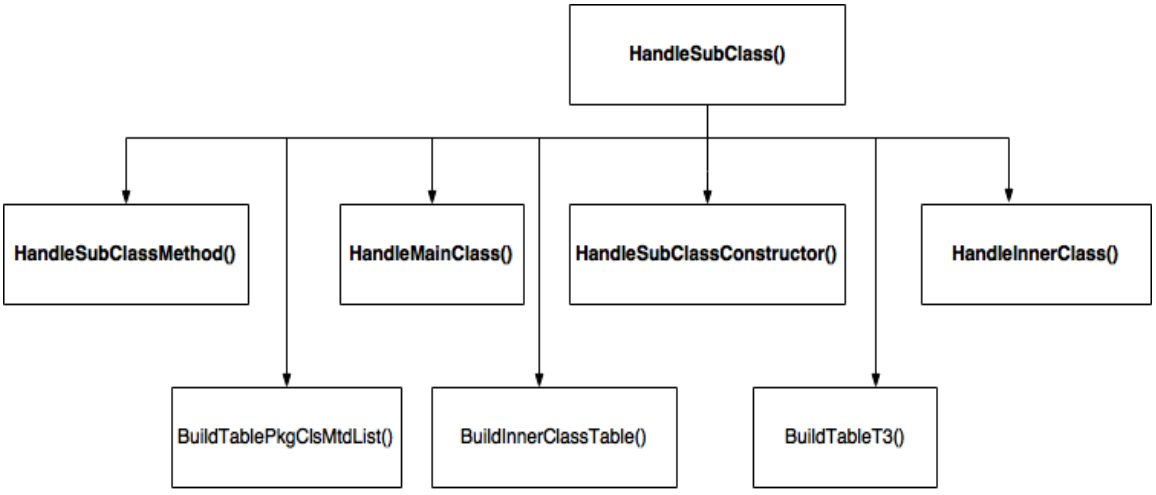


Figure 33 HandleSubClass() module

viii. HandleSubClass Module:

Usage: HandleSubClass(subclassname, srccode)

Input: SubClassname, sourcefile

Output: None

This module is handed control by CodeScanner() when it detects that a class declaration has been encountered. It is passed 2 arguments subclassname, which is the name of the class to be processed and srccode, which is the java source file name being processed.

i. The HandleSubClass() sets global and local variables for processing a sub Class.
ii. The HandleSubClass module scans the entire sub Class code in that source file. It scans line by line to handle the local variable declarations and parses the constructor/inner class and methods declared within that Class.
iii. It skips any line that is a blank or starts with a comment //. While skipping or otherwise, it always increments the global linecount by 1 when moving to the next line.
iv. When it encounters a variable declaration, it calls BuildTableT3_Var_Modifier() to store those in Table T3_Var_Modifier
v. When it encounters a constructor (a special method) it calls BuildPkgClsMtdTable() to store the tuple Pkg, Class,Method in the PkgClsMtd table and calls the HandleSubClassConstructor() module to handle further processing of a constructor declaration.
vi. When it encounters an inner class declaration, it calls the BuildInnerClassTable() to build the InnerClass Table and HandleInnerClass() for further processing of the inner class of that class.
vii. When it encounters a method declaration, it calls BuildPkgClsMtdTable() to store the tuple Pkg, Class,Method in the PkgClsMtd table and HandleSubClassMethod() module for further processing of the method.
viii. When it encounters a main class declaration, it calls the HandleMainClass() module for further processing of the main routine (dynamic analysis) in that class.

ix. Once execution is complete the module hands control back to the CodeScanner() module.

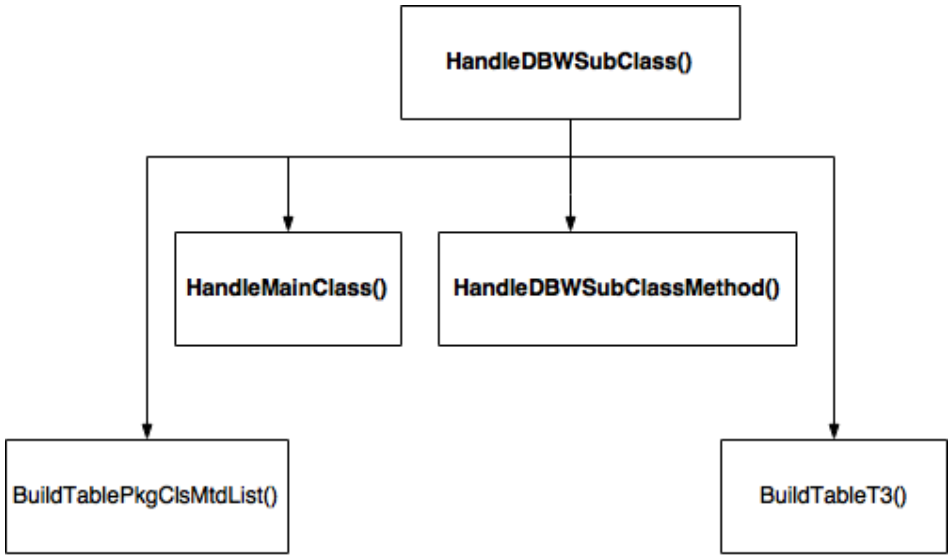


Figure 34 HandleDBWSubClass() module

ix. HandleDBWSubClass Module:

Usage: HandleDBWSubClass(DBWsubclassname, srccode)

Input: DBWSubClassname, sourcefile

Output: None

This module is handed control by CodeScanner() when it detects that a DBW sub class declaration has been encountered. It is passed 2 arguments DBWsubclassname, which is the name of the class to be processed and srccode, which is the java source file name being processed.

i. The HandleDBWSubClass() sets global and local variables for processing a DBW sub Class.

ii. The HandleDBWSubClass module scans the entire DBWsub Class code in that source file. It scans line by line to handle the local variable declarations and parses the constructor/inner class and methods declared within that Class.

iii. It skips any line that is a blank or starts with a comment //. While skipping or otherwise, it always increments the global linecount by 1 when moving to the next line.

iv. When it encounters a variable declaration, it calls BuildTableT3_Var_Modifier() to store those in Table T3_Var_Modifier

v. When it encounters a method declaration, it calls BuildPkgClsMtdTable() to store the tuple Pkg, Class,Method in the PkgClsMtd table and HandleDBWSubClassMethod() module for further processing of the method.

vi. When it encounters a main class declaration, it calls the HandleMainClass() module for further processing of the main routine (dynamic analysis) in that class.

vii. Once execution is complete the module hands control back to the CodeScanner() module.

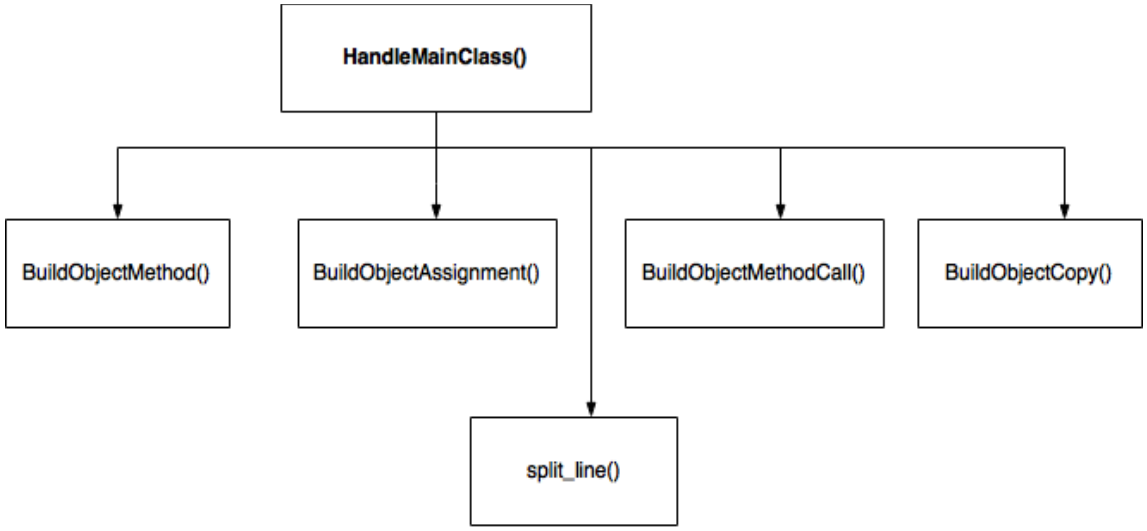


Figure 35 HandleMainClass() module

x. HandleMainClass Module:

Usage: HandleMainClass(MainClass, srcfiles)

Input: Classname, sourcefile

Output: None

This module is handed control by the HandleClass() module when it detects that the class being declared is the same as the name of the java source code file. It is passed 2 arguments. The MainClass is the name of the class with the same name as the source file and srcfiles, which is the name of the java source code file being processed.

This means that the processing has encountered the Main class declaration in the Class source code and hence calls this module for further scanning and processing. The module will set any global and local variables needed to process a main class.

i. This module writes out a message indicating that it is processing the "Dynamic Part" in the code.
ii. It skips any line that is a blank or starts with a comment //. While skipping or otherwise, it always increments the global linecount by 1 when moving to the next line.
iii. It first processes any local variables that may be declared within the class.
iv. It then goes on to process all the objects that are created by the use of the "new" keyword. If Object is of secure class it calls BuildObjectMethod() to build ObjectMethod Table
v. If the object uses a constructor method for initialization of the object and if the constructor has been identified as a secure constructor, then the module will call BuildObjectMethod() to build ObjectMethod Table.
vi. If a secure object is copied into another object, then the new object is an alias of the original object and BuildObjectCopy() is called to build the ObjectCopy Table
vii. If the secure object makes a call to a method, then a BuildObjectMethodCall() is called to build the ObjectMethodCall Table
viii. The HandleMainClass module then Scans next line and goes back to ii till all lines are processed.
ix. Once execution is complete the module hands control back to the HandleClass() module.

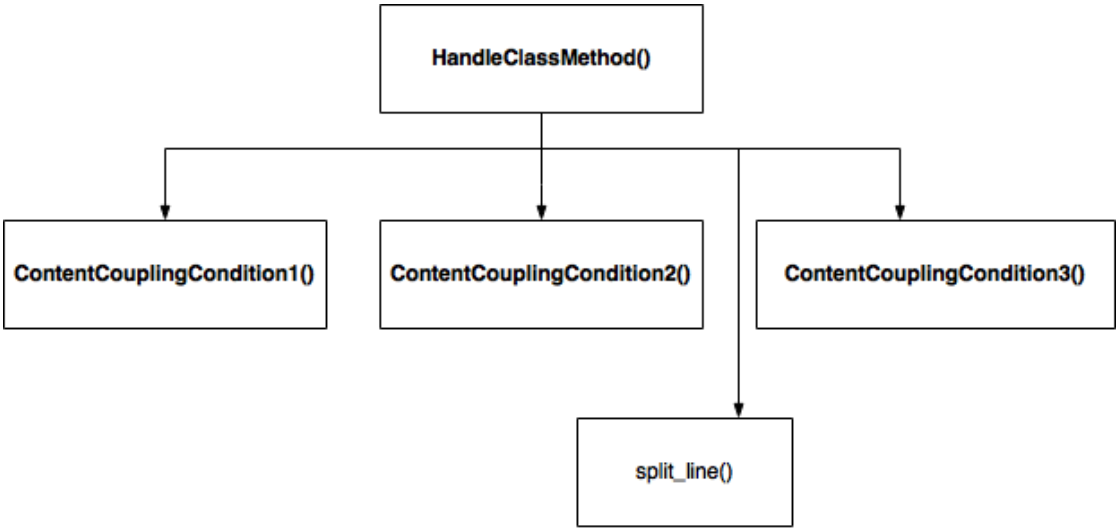


Figure 36 HandleClassMethod() module

xi. HandleClassMethod module:

Usage: HandleClassMethod(aclassname, thismethod, srcfiles)

Input: Classname, methodname, sourcefile

Output: None

HandleClassMethod() module is called by the HandleClass() module when it encounters the start of a class method within a class declaration in the java source file. It is passed 3 arguments. The aclassname is the name of the class which was currently being processed by the HandleClass Module, thismethod is the name of the method which was just encountered prior to calling this module and srcfiles, which is the name of the java source code file being processed.

i. The module sets up Boolean flags InClassMethod, ClassMethodCode and intrycatch to indicate it is in a certain processing mode.
ii. It skips any line that is a blank or starts with a comment //. While skipping or otherwise, it always increments the global linecount by 1 when moving to the next line.

iii. If a variable is assigned a value, it will call ContenCouplingCondition1() to build Table T4_ChangedVar

iv. If value from a secure variable is copied into another variable, it will call ContentCouplingCondition2() to build Table T5_UsedVar

v. If the method makes a call to a method as identified it will call ContentCouplingCondition3() to build Table T6_MethodCall

vi. End of code block detection causes the Boolean flags InClassMethod, ClassMethodCode or intrycatch to be reset once the processing is completed. It then returns control back to the HandleClass module.

vii. It will continue processing by reading next ScanLine and Jump to ii)

viii. It returns() back to the HandleClass module once the Method is processed

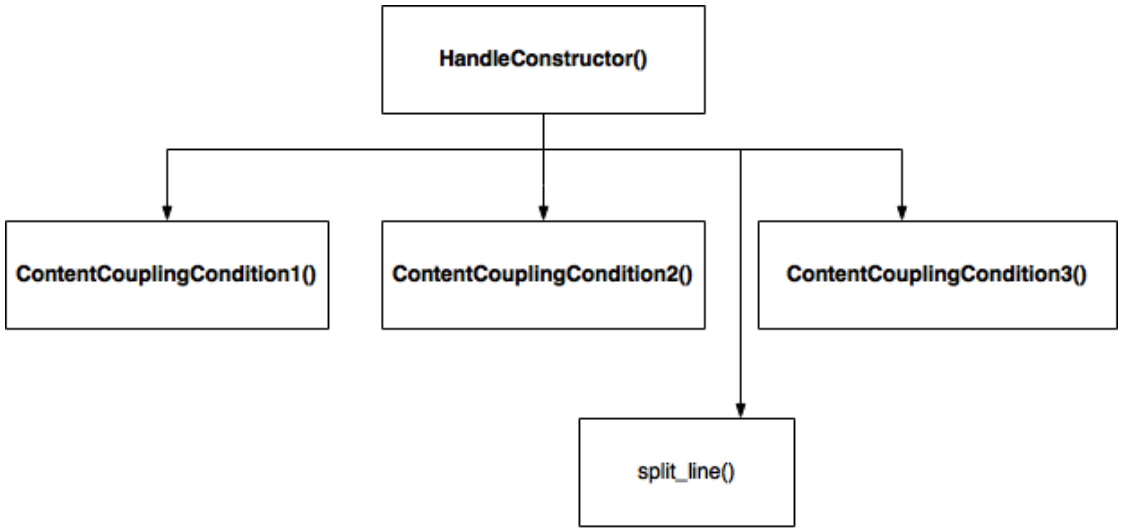


Figure 37 HandleConstructor() module

xii. HandleConstructor module:

Usage: HandleConstructor(ConstMethod, srcfiles)

Input: ConstructorMethod, sourcefile

Output: None

HandleClassMethod() module is called by the HandleClass() module when it encounters the start of a class method within a class declaration in the java source file. It is passed 3 arguments. The ConstMethod is the name of the constructor which was currently being processed by the HandleClass Module, and srcfiles, which is the name of the java source code file being processed.

i. The module sets up Boolean flags InClassMethod, ClassMethodCode and intrycatch to indicate it is in a certain processing mode.
ii. It skips any line that is a blank or starts with a comment //. While skipping or otherwise, it always increments the global linecount by 1 when moving to the next line.
iii. If a variable is assigned a value, it will call ContenCouplingCondition1() to build Table T4_ChangedVar
iv. If value from a variable is copied into another variable, it will call ContentCouplingCondition2() to build Table T5_UsedVar
v. If the method makes a call to a method as identified it will call ContentCouplingCondition3() to build Table T6_MethodCall
vi. End of code block detection causes the Boolean flags InClassMethod, ClassMethodCode or intrycatch to be reset once the processing is completed.
vii. It will continue processing by reading next ScanLine and Jump to ii)
viii. It returns() back to the HandleClass module once the Method is processed

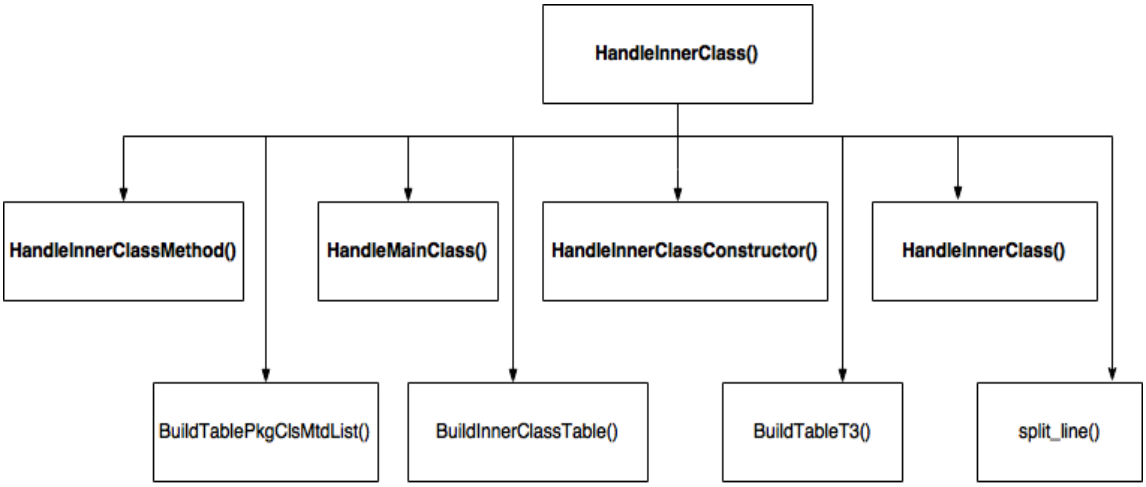


Figure 38 HandleInnerClass() module

xiii. HandleInnerClass module:

Usage: HandleInnerClass(OuterClass, InnerClass, srccode)

Input: OuterClass, InnerClass, sourcefile

Output: None

HandleInnerClass module is called by the HandleClass module when it encounters the start of an inner class declaration in the java source file. It is passed 3 arguments. The OuterClass is the name of the class which was currently being processed by the HandleClass Module , Innerclass is the name of the inner class of the outer class and srccode, which is the name of the java source code file being processed.

i. The HandleInnerClass() sets global and local variables for processing a Class.

ii. The module sets up Boolean flags InInnerClass, InnerClassCode to indicate it is in a certain processing mode.

iii. It skips any line that is a blank or starts with a comment //. While skipping or otherwise, it always increments the global linecount by 1 when moving to the next line.

iv. The HandleInnerClass module scans the entire Class code in that source file. It scans line by line to handle the local

variable declarations and also parses the constructor/inner class and methods declared within that Class.

v. When it encounters a variable declaration, it calls BuildTableT3_Var_Modifier() to store those in Table T3_Var_Modifier

vi. When it encounters a constructor (a special method) it calls BuildPkgClsMtdTable() to store the tuple Pkg, Class,Method in the PkgClsMtd table and calls the HandleInnerClassConstructor() module to handle further processing of a constructor declaration.

vii. When it encounters an inner class declaration, it calls the BuildInnerClassTable() to build the InnerClass Table and HandleInnerClass() for further processing of the inner class of that class.

viii. When it encounters a method declaration, it calls BuildPkgClsMtdTable() to store the tuple Pkg, Class,Method in the PkgClsMtd table and HandleInnerClassMethod() module for further processing of the method.

ix. When it encounters a main class declaration, it calls the HandleMainClass() module for further processing of the main routine (dynamic analysis) in that class.

x. Once execution is complete the module hands control back to the HandleClass() module.

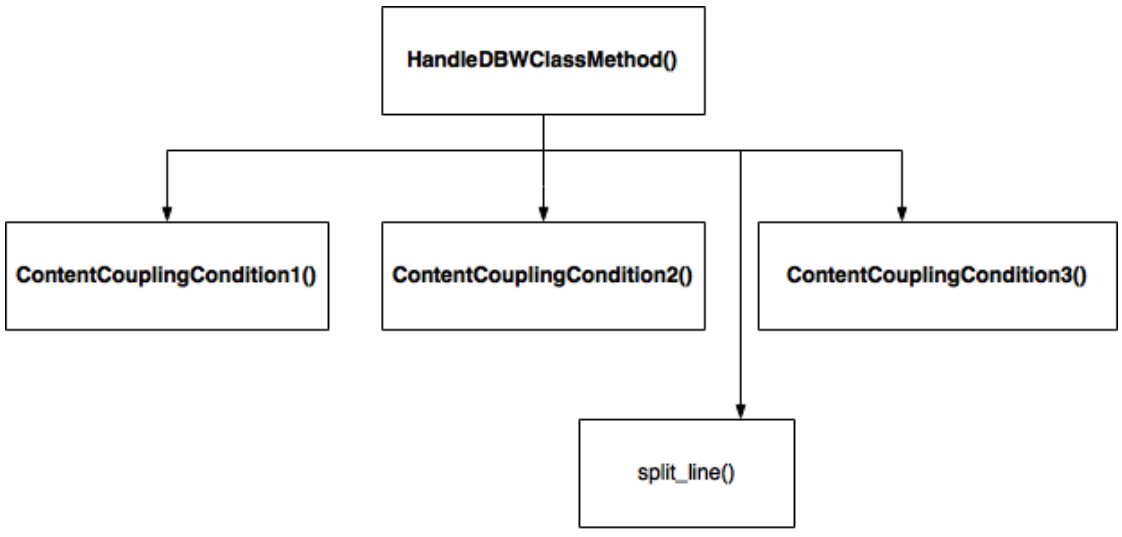


Figure 39 HandleDBWClassMethod() module

xiv. HandleDBWClassMethod module:

Usage: HandleDBWClassMethod(aclassname, thismethod, srcfiles)

Input: Classname, methodname, sourcefile

Output: None

HandleDBWClassMethod() module is called by the HandleDBWClass() module when it encounters the start of a class method within a class declaration in the java source file. It is passed 3 arguments. The aclassname is the name of the class which was currently being processed by the HandleDBWClass Module, thismethod is the name of the method which was just encountered prior to calling this module and srcfiles, which is the name of the java source code file being processed.

i. The module sets up Boolean flags InDBWClassMethod, DBWClassMethodCode, inforwhile and intrycatch to indicate it is in a certain processing mode.
ii. It skips any line that is a blank or starts with a comment //. While skipping or otherwise, it always increments the global linecount by 1 when moving to the next line.
iii. If a variable is assigned a value, it will call ContenCouplingCondition1() to build Table T4_ChangedVar
iv. If value from a secure variable is copied into another variable, it will call ContentCouplingCondition2() to build Table T5_UsedVar
v. If the method makes a call to a method as identified it will call ContentCouplingCondition3() to build Table T6_MethodCall
vi. End of code block detection causes the Boolean flags InDBWClassMethod, DBWClassMethodCode or intrycatch to be reset once the processing is completed.
vii. It will continue processing by reading next ScanLine and Jump to ii)
viii. It returns() back to the HandleDBWClass module once the Method is processed

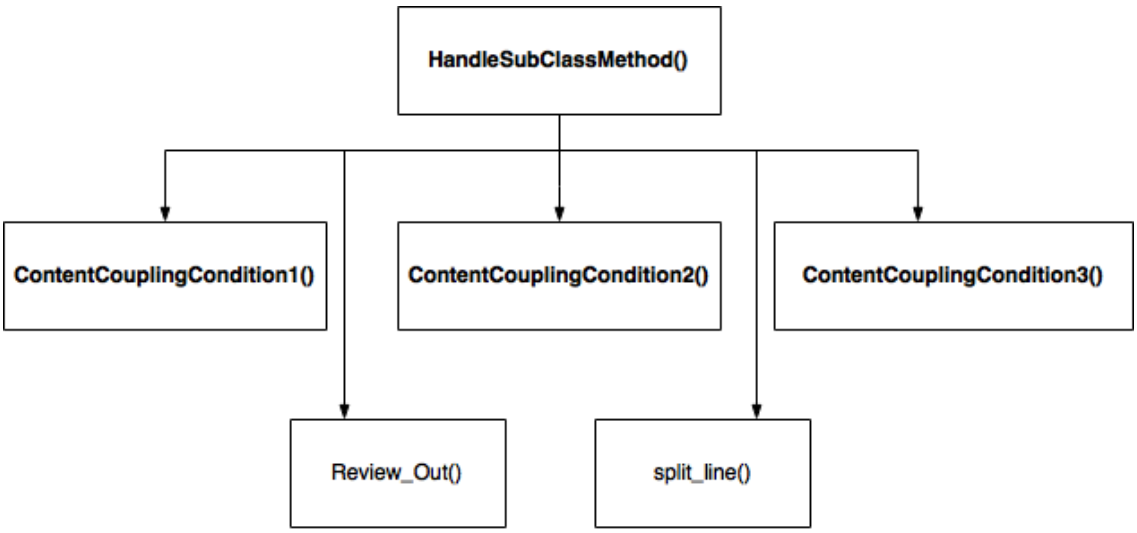


Figure 40 HandleSubClassMethod() module

xv. HandleSubClassMethod module:

Usage: HandleSubClassMethod(asubclassname, thismethod, srcfiles)

Input: Classname, methodname, sourcefile

Output: None

HandleSubClassMethod() module is called by the HandleSubClass() module when it encounters the start of a sub class method within a class declaration in the java source file. It is passed 3 arguments. The aclassname is the name of the class which was currently being processed by the HandleSubClass Module, thismethod is the name of the method which was just encountered prior to calling this module and srcfiles, which is the name of the java source code file being processed.

i. The module sets up Boolean flags InSubClassMethod, SubClassMethodCode and intrycatch to indicate it is in a certain processing mode.

ii. It skips any line that is a blank or starts with a comment //. While skipping or otherwise, it always increments the global linecount by 1 when moving to the next line.

iii. If a variable is assigned a value, it will call ContenCouplingCondition1() to build Table T4_ChangedVar

iv. If value from a secure variable is copied into another variable, it will call ContentCouplingCondition2() to build Table T5_UsedVar

v. If the method makes a call to a method as identified it will call ContentCouplingCondition3() to build Table T6_MethodCall

vi. End of code block detection causes the Boolean flags InSubClassMethod, SubClassMethodCode or intrycatch to be reset once the processing is completed.

vii. It will continue processing by reading next ScanLine and Jump to ii)

viii. It returns() back to the HandleSubClass module once the Method is processed

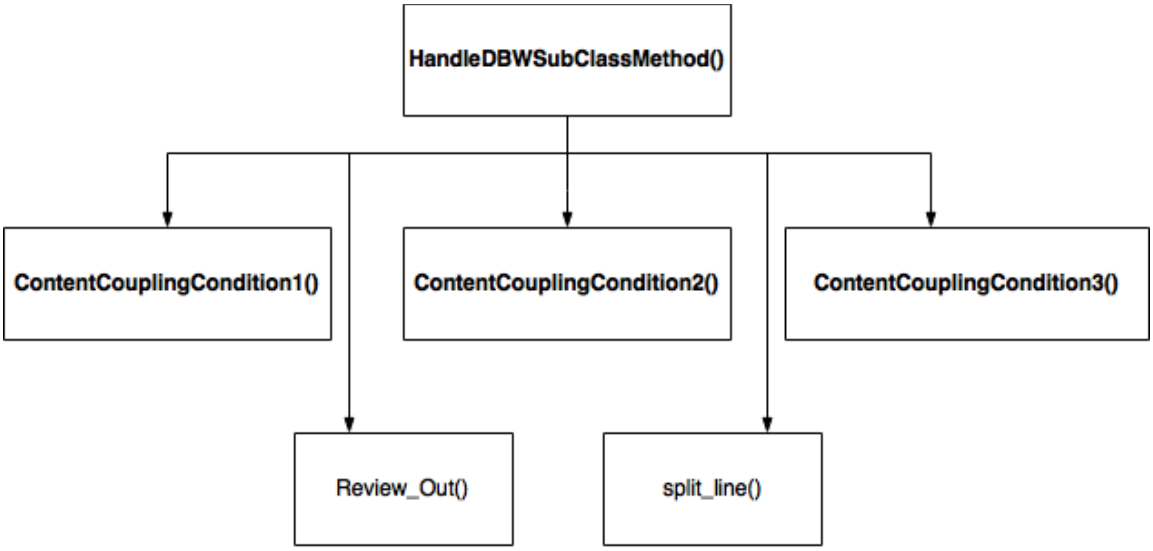


Figure 41 HandleDBWSubClassMethod()

xvi. HandleDBWSubClassMethod module:

Usage: HandleDBWSubClassMethod(asubclassname, thismethod, srcfiles)

Input: Classname, methodname, sourcefile

Output: None

HandleDBWSubClassMethod() module is called by the HandleDBWSubClass() module when it encounters the start of a sub class method within a class declaration in the java source file. It is passed 3 arguments. The aclassname is the name of the class which was currently being processed by the HandleDBWSubClass Module, thismethod is the name of the method which was just encountered prior to calling this module and srcfiles, which is the name of the java source code file being processed.

i. The module sets up Boolean flags InDBWSubClassMethod, DBWSubClassMethodCode and intrycatch to indicate it is in a certain processing mode.
ii. It skips any line that is a blank or starts with a comment //. While skipping or otherwise, it always increments the global linecount by 1 when moving to the next line.
iii. If a variable is assigned a value, it will call ContenCouplingCondition1() to build Table T4_ChangedVar
iv. If value from a secure variable is copied into another variable, it will call ContentCouplingCondition2() to build Table T5_UsedVar
v. If the method makes a call to a method as identified it will call ContentCouplingCondition3() to build Table T6_MethodCall
vi. End of code block detection causes the Boolean flags InDBWSubClassMethod, DBWSubClassMethodCode or intrycatch to be reset once the processing is completed.
vii. It will continue processing by reading next ScanLine and Jump to ii)
viii. It returns() back to the HandleDBWSubClass module once the Method is processed

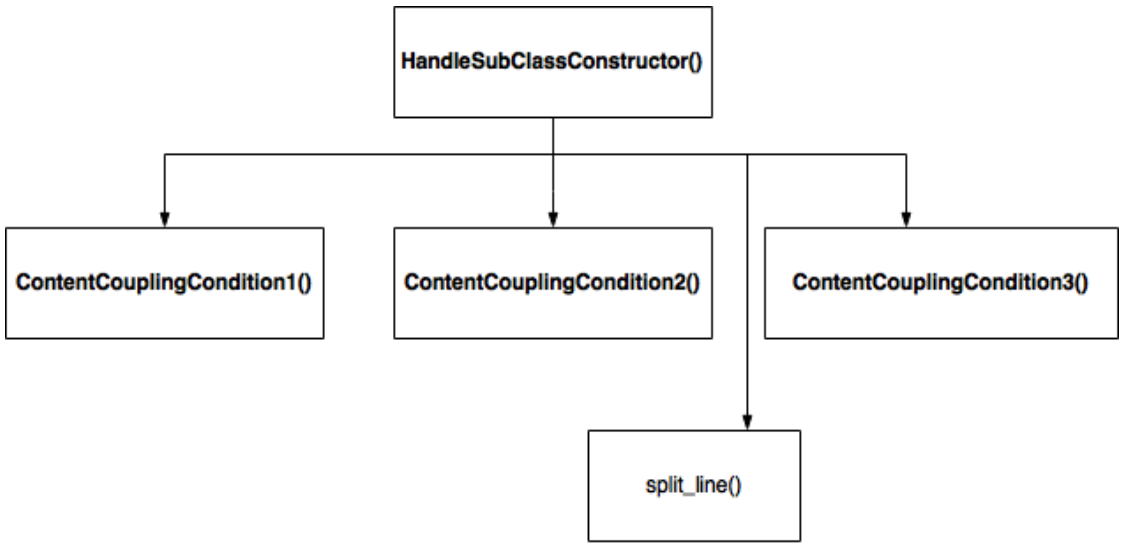


Figure 42 HandleSubClassConstructor() module

xvii. HandleSubClassConstructor module:

Usage: HandleSubClassConstructor(SubConstMethod, srcfiles)

Input: SubclassConstructorMethod, sourcefile

Output: None

HandleSubClassConstructor() module is called by the HandleSubClass() module when it encounters the start of a sub class method within a class declaration in the java source file. It is passed 2 arguments. The SubConstMethod is the name of the subclass which was currently being processed by the HandleSubClass Module, and srcfiles, which is the name of the java source code file being processed.

i. The module sets up Boolean flags InSubClassMethod, SubClassMethodCode and intrycatch to indicate it is in a certain processing mode.
ii. It skips any line that is a blank or starts with a comment //. While skipping or otherwise, it always increments the global linecount by 1 when moving to the next line.
iii. If a variable is assigned a value, it will call ContenCouplingCondition1() to build Table T4_ChangedVar

iv. If value from a secure variable is copied into another variable, it will call ContentCouplingCondition2() to build Table T5_UsedVar

v. If the method makes a call to a method as identified it will call ContentCouplingCondition3() to build Table T6_MethodCall

vi. End of code block detection causes the Boolean flags InSubClassMethod, SubClassMethodCode or intrycatch to be reset once the processing is completed.

vii. It will continue processing by reading next ScanLine and Jump to ii)

viii. It returns() back to the HandleSubClass module once the Method is processed

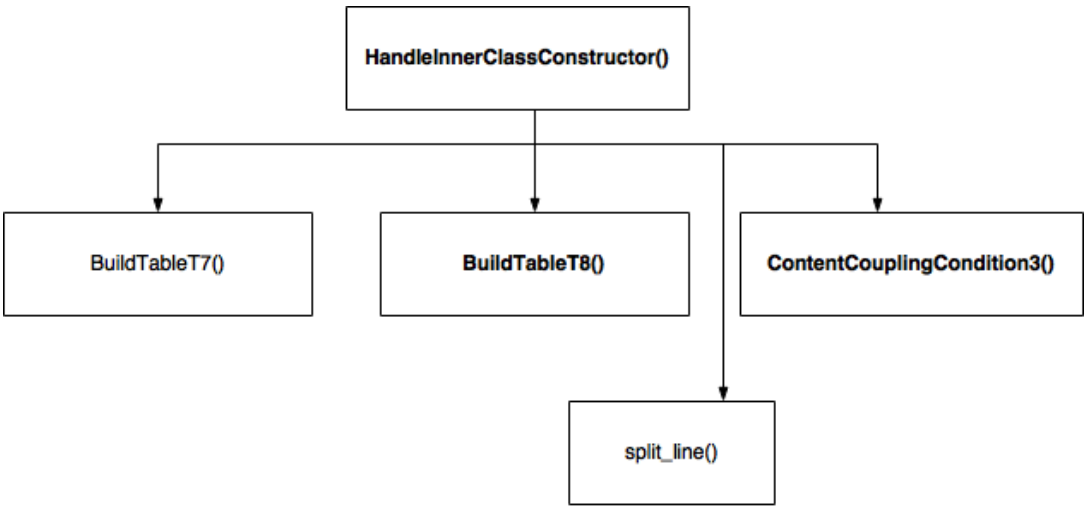


Figure 43 HandleInnerClassConstructor() module

xviii. HandleInnerClassConstructor module:

Usage: HandleInnerClassConstructor(Outer, InnerConstMethod, srcfiles)

Input: OuterClass, InnerClassConstructorMethod, sourcefile

Output: None

HandleInnerClassConstructor() module is called by the HandleInnerClass() module when it encounters the start of a class method within a class declaration in the java source file. It is passed 3 arguments. The Outer is the Outer class of the innerclass that calls this

method. The InnerConstMethod is the name of the constructor which was currently being processed by the HandleInnerClass Module, and srcfiles, which is the name of the java source code file being processed.

i. The module sets up Boolean flags InInnerClassMethod, InnerClassMethodCode and intrycatch to indicate it is in a certain processing mode.
ii. It skips any line that is a blank or starts with a comment //. While skipping or otherwise, it always increments the global linecount by 1 when moving to the next line.
iii. If a variable is assigned a value, it will call BuildTableT7_InnClsChgdVar() to build Table T7_InnClsChgdVar
iv. If value from a variable is copied into another variable, it will call BuildTableT8_InnClsUsedVar() to build Table T8_InnClsUsedVar
v. If the constructor makes a call to a method as identified it will call ContentCouplingCondition3() to build Table T6_MethodCall
vi. End of code block detection causes the Boolean flags InInnerClassMethod, InnerClassMethodCode or intrycatch to be reset once the processing is completed.
vii. It will continue processing by reading next ScanLine and Jump to ii)
viii. It returns() back to the HandleInnerClass module once the constructor is processed

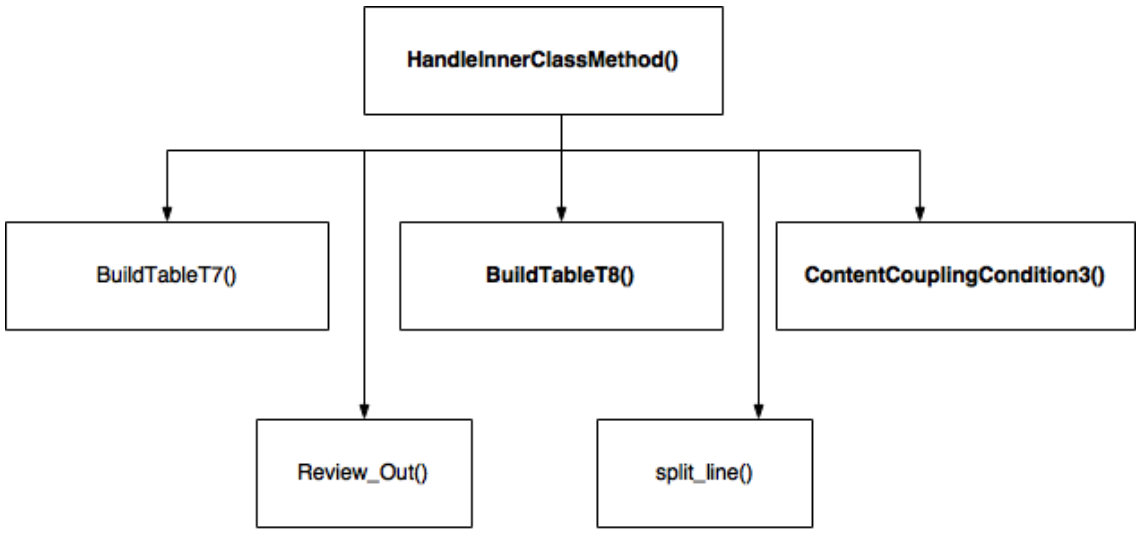


Figure 44 HandleInnerClassMethod() module

xix. HandleInnerClassMethod module:

Usage: HandleInnerClassMethod(aclassname, InnerClass, thismethod, srcfiles)

Input: Classname, InnerClass, methodname, sourcefile

Output: None

HandleInnerClassMethod() module is called by the HandleInnerClass() module when it encounters the start of a class method within a class declaration in the java source file. It is passed 4 arguments. The aclassname is the name of the outer class of the inner class InnerClass. The thisMethod is the name of the method which was currently being processed by the HandleInnerClass Module, and srcfiles, which is the name of the java source code file being processed.

i. The module sets up Boolean flags InInnerClassMethod, InnerClassMethodCode and intrycatch to indicate it is in a certain processing mode.
ii. It skips any line that is a blank or starts with a comment //. While skipping or otherwise, it always increments the global linecount by 1 when moving to the next line.
iii. If a variable is assigned a value, it will call BuildTableT7_InnClsChgdVar() to build Table T7_InnClsChgdVar
iv. If value from a variable is copied into another variable, it will call BuildTableT8_InnClsUsedVar() to build Table T8_InnClsUsedVar
v. If the constructor makes a call to a method as identified it will call ContentCouplingCondition3() to build Table T6_MethodCall
vi. End of code block detection causes the Boolean flags InInnerClassMethod, InnerClassMethodCode or intrycatch to be reset once the processing is completed.
vii. It will continue processing by reading next ScanLine and Jump to ii)
viii. It returns() back to the HandleInnerClass module once the constructor is processed

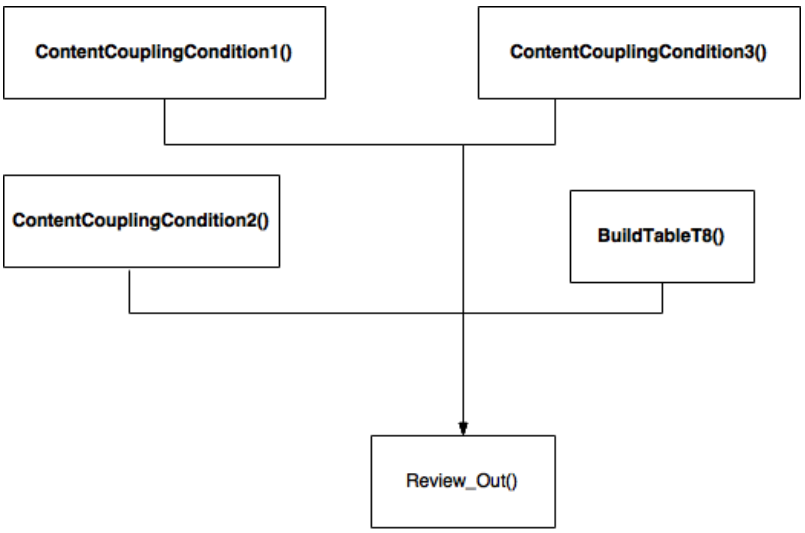


Figure 45 Low level modules calling Review_Out()

xx. Low level modules

The ContentCouplingCondition1() module builds the Table T4_ChangedVar

The ContentCouplingCondition2() module builds the Table T5_UsedVar

The ContentCouplingCondition3() module builds the Table T6_MethodCall

There are numerous other small modules that help in splitting tokens before a (, after a (, split the scanline into tokens, split before or after a "." Etc. All of those may not be listed in this design.

1. SplitBeforeRound module:

Usage: SplitBeforeRound(mixword)

Input: mixword

Output: String before the "(" In mixword is returned

i. SplitBeforeRound module is passed mixword as an argument.
ii. The mixword contains a left parenthesis (in the string and the module simply splits the mixword into 2 and returns the part of the string before the "(" to the calling routine.

2. SplitSAfterRound module:

Usage: SplitAfterRound(mixword)

Input: mixword

Output: String after the "(" In mixword is returned

i. SplitAfterRound module is passed mixword as an argument.
ii. The mixword contains a left parenthesis (in the string and the module simply splits the mixword into 2 and returns the part of the string after the "(" back to the calling routine.

3. SplitBeforeDot module:

Usage: SplitBeforeDot(mixword)

Input: mixword

Output: String before the "." In mixword is returned

i. SplitBeforeDot module is passed mixword as an argument.
ii. The mixword contains a period "." in the string and the module simply splits the mixword into 2 and returns the part of the string before the "." to the calling routine.

4. SplitAfterDot module:

Usage: SplitAfterDot(mixword)

Input: mixword

Output: String after the "." is returned

i. SplitAfterDot module is passed mixword as an argument.
ii. The mixword contains a period "." in the string and the module simply splits the mixword into 2 and returns the part of the string after the "." back to the calling routine.

5. GetArguments module:

Usage: GetArguments(largeword)

Input: largeword

Output: List of arguments is returned

i. GetArguments module is passed largeword, a string, as an argument. The largeword contains a set of arguments to a function enclosed within parentheses.
ii. The module simply splits the largeword into a list of arguments and returns splitword (the part of the string before the left parenthesis "(") and arguments back to the calling routine.

6. Review_Out() module:

Usage: Review_Out(roclassname,ronewstring,rostring,rolinecount,rovartype)

Input: roclassname, ronewstring, rostring, rolinecount, rovartype

Output: None

i. Review_Out module is passed classname, string1, string2, line num and variable type as the arguments. It prints out information to the Output file for the user to review after the end of the Scanning and analyzing process.
ii. The module simply writes the information to the outputfile and returns to the calling routine.

7. split_line() module:

Usage: SplitLine(myline)

Input: ScanLine

Output: List of tokens from ScanLine

i. split_line module is passed the Scanned Line from a java source file as the argument.
ii. It splits the line into tokens ignoring a list of symbols. { ; [ ] . ( ) { } = + - * ^ / " }
iii. The module returns a list of tokens to the calling routine.

8. split_words() module:

Usage: split_words(myline)

Input: ScanLine

Output: List of tokens from ScanLine

i. split_words module is passed the Scanned Line from a java source file as the argument.
ii. It splits the line into tokens ignoring a list of symbols. { ; [ ] . ( ) { } = “ }
iii. The module returns a list of tokens to the calling routine.

APPENDIX B - SAMPLE OUTPUT FROM CASE STUDY

1) Static Analysis – Record of All Methods – Sample Output

```
========= Starting the Scanning and Analyzer process ===========

1] - RECORD OF ALL METHODS
============================

Total number of methods = 40

OBSCaseStudy    -Account         -Account
OBSCaseStudy    -Account         -getBalance
OBSCaseStudy    -Account         -credit
OBSCaseStudy    -Account         -debit
OBSCaseStudy    -Account         -getAccountNumber
OBSCaseStudy    -AccountDB       -getAccount
OBSCaseStudy    -AccountDB       -getAccountNumbers
OBSCaseStudy    -AccountDB       -getBalances
OBSCaseStudy    -AccountDB       -credits
OBSCaseStudy    -AccountDB       -debits
OBSCaseStudy    -CheckingAccount -CheckingAccount
OBSCaseStudy    -Customer        -Customer
OBSCaseStudy    -Customer        -getCustName
OBSCaseStudy    -Customer        -getCustAddress
OBSCaseStudy    -Customer        -validateCPIN
OBSCaseStudy    -CustomerDB      -getCustomer
OBSCaseStudy    -CustomerDB      -authenticateUsers
OBSCaseStudy    -CustomerDB      -getCustomerInfo
OBSCaseStudy    -Deposit         -Deposit
OBSCaseStudy    -Deposit         -executeDeposit
OBSCaseStudy    -Deposit         -promptForDepositAmount
OBSCaseStudy    -OBS             -OBS
OBSCaseStudy    -OBS             -run
OBSCaseStudy    -OBS             -authenticateUser
…
…
OBSCaseStudy    -Transfer        -displayMenuOfTransferAmounts
OBSCaseStudy    -Withdrawal      -Withdrawal
OBSCaseStudy    -Withdrawal      -executeWithdrawal
OBSCaseStudy    -Withdrawal      -displayMenuOfAmounts

END OF METHODS LIST OUTPUT
============================
```

2) Record of CR Methods including duplicates – Sample Output

```
2] - RECORD OF CR METHODS (Includes Duplicates!)
================================================

Total number of Records including duplicates = 106

OBSCaseStudy    -Query          -executeQuery         --6-3A
OBSCaseStudy    -Account        -getAccountNumber     --6-3B
OBSCaseStudy    -Account        -getBalance           --6-3B
OBSCaseStudy    -TransactionLogDB-transactionlog      --6-3B
OBSCaseStudy    -CustomerDB     -authenticateUsers    --6-3B
OBSCaseStudy    -CustomerDB     -getCustomerInfo      --6-3B
OBSCaseStudy    -Account        -Account        --6-1A/--6-2A
OBSCaseStudy    -Account        -debit          --6-1A/--6-2A
OBSCaseStudy    -AccountDB      -getAccountNumbers--6-1A/--6-2A
OBSCaseStudy    -AccountDB      -getBalances    --6-1A/--6-2A
OBSCaseStudy    -AccountDB      -credits   --6-1A/--6-2A
…
OBSCaseStudy    -Account        -credit               --6-1B
OBSCaseStudy    -Account        -debit                --6-1B
OBSCaseStudy    -AccountDB      -getAccount     --6-1D/--6-2D
OBSCaseStudy    -TransactionLogDB-transactionlog --6-2D/--6-3E
OBSCaseStudy    -AccountDB      -getBalances    --6-2D/--6-3E
OBSCaseStudy    -TransactionLogDB-transactionlog --6-2D/--6-3E
OBSCaseStudy    -AccountDB      -getAccountNumbers--6-1C/--6-2C2
OBSCaseStudy    -AccountDB      -getAccountNumbers    --6-3A
OBSCaseStudy    -AccountDB      -getBalances    --6-1C/--6-2C2
OBSCaseStudy    -AccountDB      -credits              --6-3A
OBSCaseStudy    -AccountDB      -debits         --6-1C/--6-2C2
OBSCaseStudy    -AccountDB      -debits               --6-3A
OBSCaseStudy    -CustomerDB     -authenticateUsers--6-1C/--6-2C2
OBSCaseStudy    -CustomerDB     -authenticateUsers    --6-3A
OBSCaseStudy    -CustomerDB     -getCustomerInfo--6-1C/--6-2C2
OBSCaseStudy    -CustomerDB     -getCustomerInfo      --6-3A
OBSCaseStudy    -Deposit        -executeDeposit  --6-1C/--6-2C2
OBSCaseStudy    -Deposit        -executeDeposit       --6-3A
OBSCaseStudy    -OBS            -run            --6-1C/--6-2C2
OBSCaseStudy    -OBS            -authenticateUser     --6-3A
OBSCaseStudy    -OBS       -performTransactions--6-1C/--6-2C2
OBSCaseStudy    -OBS       -performTransactions  --6-3A

END OF CR METHODS LIST OUTPUT
=============================
```

3) Reason for Static Security hotspots – Sample Output

```
3 a] Reasons for Variable Hot Spot determination
================================================
OBSCaseStudy-Account-accountNumber - This is Secure variable
declaration at line number 6

OBSCaseStudy-Account-CustID - This is Secure variable declaration
at line number 8

OBSCaseStudy-AccountDB-conn - This is variable declaration at
line number 8
WARNING! Please change variable declaration to private or
protected before any further work

OBSCaseStudy-AccountDB-stmt - This is variable declaration at
line number 9
WARNING! Please change variable declaration to private or
protected before any further work
…
…
3 b] Reasons for CR determination
=================================

Total number of CR determination records = 106

OBSCaseStudy-Query-executeQuery Warning: 6-3A Non Secure Method
executeQuery calls Secure method getBalances in other class,
please review new CR method pkg OBSCaseStudy,class Query, and
method executeQuery at line number 27

OBSCaseStudy-Account-getAccountNumber Warning: 6-3B Secure Method
getAccountNumbers from class AccountDB calls non secure method
getAccountNumber in other class, please review new CR method pkg
OBSCaseStudy,class Account, and method getAccountNumber at line
number 74

OBSCaseStudy-CustomerDB-getCustomer Warning: 6-1D/--6-2D Secure
Method authenticateUsers calls non secure method getCustomer in
same secure class, please review new CR method pkg
OBSCaseStudy,class CustomerDB, and method getCustomer at line
number 57
```

4) Record of Unique CR Methods – Sample Output

```
4] - RECORD OF Unique CR METHODS
==========================

Total number of Unique CR methods = 34

OBSCaseStudy-Customer-Customer
OBSCaseStudy-Deposit-Deposit
OBSCaseStudy-OBS-OBS
OBSCaseStudy-Transfer-Transfer
OBSCaseStudy-Withdrawal-Withdrawal
OBSCaseStudy-AccountDB-getAccount
OBSCaseStudy-CustomerDB-getCustomer
OBSCaseStudy-Deposit-promptForDepositAmount
OBSCaseStudy-OBS-SelectAccountType
OBSCaseStudy-OBS-displayMainMenu
OBSCaseStudy-Transfer-displayMenuOfTransferAmounts
OBSCaseStudy-Withdrawal-displayMenuOfAmounts
OBSCaseStudy-Account-Account
OBSCaseStudy-Account-getAccountNumber
OBSCaseStudy-Account-getBalance
OBSCaseStudy-Account-credit
OBSCaseStudy-Account-debit
…
…
OBSCaseStudy-Customer-getCustPhone
OBSCaseStudy-TransactionLogDB-displaytransactionlog
OBSCaseStudy-TransactionLogDB-transactionlog
OBSCaseStudy-AccountDB-getAccountNumbers
OBSCaseStudy-Deposit-executeDeposit
OBSCaseStudy-OBS-run
OBSCaseStudy-OBS-authenticateUser
OBSCaseStudy-OBS-GetAccount
OBSCaseStudy-OBS-performTransactions
OBSCaseStudy-Query-executeQuery
OBSCaseStudy-Transfer-executeTransfer
OBSCaseStudy-Withdrawal-executeWithdrawal

END OF Unique CR METHODS LIST OUTPUT

Please use the DB Browser for SQLite to view CR Methods table to
understand the reasons why each method is a CR method
============================
```

5) Results of Dynamic Analysis – Sample Output

```
5] Results of Dynamic Analysis
==========================

Total number of dynamic analysis records = 28

OBSCaseStudy-OBSClient-main Warning: Dynamic Rule1 - New secure
object cust of secure class Customer created in source file
CustomerDB at line number 37

OBSCaseStudy-OBSClient-main Warning: Dynamic Rule1_2 - New secure
object cust of class Customer calls secure constructor Customer
in source file CustomerDB at line number 37

OBSCaseStudy-OBSClient-main Warning: Dynamic Rule1 - New secure
object TTransaction of secure class Transfer created in source
file OBS at line number 155

OBSCaseStudy-OBSClient-main Warning: Dynamic Rule1_2 - New secure
object TTransaction of class Transfer calls secure constructor
Transfer in source file OBS at line number 155

…
…
…
…

OBSCaseStudy-OBSClient-main Warning: Dynamic Rule2 - Object
TTransaction of class Transfer calls secure method
displayMenuOfTransferAmounts in source file Transfer at line
number 44

OBSCaseStudy-OBSClient-main Warning: Dynamic Rule2 - Object
DTransaction of class Deposit calls secure method executeDeposit
in source file OBS at line number 166

OBSCaseStudy-OBSClient-main Warning: Dynamic Rule2 - Object
WTransaction of class Withdrawal calls secure method
executeWithdrawal in source file OBS at line number 175

END OF Dynamic Analysis
============================

========== End of the Scanner and Analyzer process ============
```

INDEX