SECURITY-AWARE PROGRAM VISUALIZATION FOR ANALYZING IN-LINED REFERENCE MONITORS

by

Aditi A. Patwardhan

APPROVED BY SUPERVISORY COMMITTEE:

_____

Dr. Kevin Hamlen, Chair

_____

Dr. Kendra Cooper

_____

Dr. Murat Kantarcioglu

SECURITY-AWARE PROGRAM VISUALIZATION FOR ANALYZING IN-LINED REFERENCE MONITORS

by

ADITI A. PATWARDHAN, B.S.

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

August, 2010

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Dr Hamlen and Dr Cooper for accepting me as their student, initially for an independent study project and later to continue it as a thesis. I thank them both for the numerous reviews and constant feedback, especially with the documentation work. I thank Dr Hamlen for being my advisor and for guiding me through all the challenges that I faced while completing this thesis. I thank Dr Cooper for continually motivating me towards achieving greater quality.

I would like to say a special thank-you to Dr Murat for serving as a committee member and extending his support for the thesis defense.

I thank Jeannette Bennett for the tool developed for her thesis which served as an initial implementation and technology idea for the visualizer tool. I am grateful to the Language-Based Security group and I learnt a lot by participating in the research meetings.

I express great gratitude towards my parents Swati and Abhijit Patwardhan and my fiancé Ashutosh Watway, for their unconditional love and support. I will always attribute my success and achievements in my life to them. Lastly I acknowledge all my friends and roommates here at UTD for being my constant emotional support throughout my graduate studies away from home.

June, 2010

SECURITY-AWARE PROGRAM VISUALIZATION FOR ANALYZING IN-LINED REFERENCE MONITORS

Publication No. _____

Aditi A. Patwardhan, M.S.
The University of Texas at Dallas, 2010

Supervising Professor: Dr. Kevin Hamlen

In-lined Reference Monitoring frameworks are an emerging technology for enforcing security policies over untrusted, mobile, binary code. However, formulating correct policy specifications for such frameworks to enforce remains a daunting undertaking with few supporting tools. A visualization approach is proposed to aid in this task. In contrast to existing approaches, which typically involve tedious and error-prone manual inspection of complex binary code, the proposed framework provides automatically generated, security-aware visual models that follow the UML specification. This facilitates formulation and testing of prototype security policy specifications in a faster and more reliable manner than is possible with existing manual approaches.

TABLE OF CONTENTS

LIST OF FIGURES

Number Page

LIST OF TABLES

Number Page

CHAPTER 1

INTRODUCTION

1.1 Motivation

Software security is becoming increasingly important with the growth of the Internet and mobile code technologies like Java. Mobile code technologies generate software components for environments in which code-consumers receive code from separate code-producers. These software components are mainly distributed as binary executable files that are downloaded from web pages or as email attachments. In many realistic settings, not all code-producers are fully trusted; for example, webpages may be served from untrusted servers or emails may arrive from untrusted senders. Security is an obvious concern in such an environment where binary, executable mobile code is received over the Internet, whose code-producer is not known or is not fully trusted. Malicious adversaries can cause security attacks in the form of malware that is distributed over the Internet as mobile code. Every year billions of dollars are invested by business enterprises to protect against or recover from such software security attacks [1]. Security violations range from information leakage to access control violations and data corruption. Given the high financial costs, software security concerns have become increasingly critical for many business environments.

1.2 Software Security Background

To ensure a secure mobile code environment, we would typically want to define security policies that specify certain constraints on software behavior, and then enforce these

policies on any untrusted software to be executed on the system. Specifically, security policies define trace properties of a program execution. These include liveness properties, which specify that some “good event” should eventually happen; for example a process should eventually release a lock that it has acquired during its execution. And they include safety properties, which specify that some “bad event” should not happen; for example, a process should not attempt to modify system executable files. Precisely enforcing arbitrary liveness policies at the software level remains an open challenge [2], but most practical policies can be reformulated as safety policies. For example, time-bounded policies, such as the policy that prohibits a process from holding a lock for more than 1000 instruction cycles, are safety policies [3]. Hence, we focus on the class of security policies that are safety properties.

A classic example for a safety policy is the access control policy that prohibits writes to files for which the user does not have write permission and prohibits reads from files for which the user does not have read permission. Such a security policy defines these unauthorized accesses as “bad” events. The definition of a “bad” event can also rely on the past history of the execution. An example of such a history-based policy is to avoid information leakage over the network by prohibiting a process from sending any information over the network once it has read a confidential file. A resource bound policy could dictate that the application should not open more than 100 files in its lifetime, to avoid resource exhaustion. An auditing policy could ensure that any security relevant operation should be logged before the next security relevant operation is executed. A peer-to-peer network might stipulate that any user may download at most 2 files more than the number of uploads made by the user, hence preventing the occurrence of ‘free-riding’, see [6].

Safety policies that are history-based (i.e., stateful) can be encoded as security automata. A security automaton is a finite state automaton that accepts policy-permitted sequences of security-relevant events [3]. The security automaton makes transitions on program operations that constitute security-relevant events. If a security-relevant event is encountered that causes a policy violation, the automaton rejects the corresponding execution sequence. Bisimulation of the security automaton with the untrusted program is used as a mechanism to enforce the underlying security policies. When a security violation is detected by the automaton, the corresponding execution is terminated. This approach is demonstrated by in-lined reference monitors (IRM's) [3].

The IRM systems are one of the current state-of-art enforcement mechanisms that monitor the program execution for any security relevant event that occurs and halts the execution (or takes some other corrective action) if an impending security violation is detected. The SASI system [4] implements these for x86 assembly code and Java bytecode architectures. The Java-MOP system [5] works with the aspect-oriented programming paradigm. It specifies the desired security properties, along with the code to execute if a security violation is detected. The specification is then translated to AspectJ code and integrated into the application program using an aspect weaver. The specification is written as a combination of linear temporal logic (LTL) and trusted code fragments [5]. The SPoX (Security Policy XML) system [6] provides a purely declarative policy specification language in which security-relevant events are designated via AspectJ pointcuts and policies over these events are specified as security state transitions [6]. A SPoX specification denotes a (finite or infinite state) security automaton that makes transitions on security relevant events and rejects the execution sequence if a security violation is detected.

1.3 Research Problem

IRM's provide a powerful means to enforce application-specific security policies, but identifying and defining a good security policy usually requires a fairly deep understanding the underlying application code. Since in many contexts at least some components of untrusted software are not available as source code, this often requires the policy writer to analyze the underlying application structure and control-flows.

For example, to prohibit network-send operations, one must be able to rigorously define what constitutes a network-send operation at the binary level. In architectures with complex runtime systems, such as Java, there may be hundreds of primitive instructions that constitute security-relevant operations, each of which must be identified in a complete specification of the policy. If the IRM signals unexpected policy violations for non-malicious applications during testing, the policy-writer must understand which operations may have been misidentified as policy-violating by the flawed specification. In general, much manual analysis and inspection of malicious and non-malicious binary code may be required in order to formulate a policy that prohibits all undesired program behaviors without curtailing desired behaviors.

This manual task becomes very tedious and error-prone as the size of the binary code increases. A visualization tool is needed to aid the analysis of the untrusted binary code and to facilitate faster and more reliable discovery and prototyping of application-specific security policies.

1.4 Proposed Solution

We present a Visualization Framework that generates security-aware visual models for the low-level Java bytecode. The static visual model represents the underlying class

structures; the dynamic visual model represents the possible execution sequences (control flows) in the application. We choose the Unified Modeling Language (UML) specification for our visual model. The UML has become the de facto standard for visual modeling of software applications [8]. We represent the class structures and their relationships using the UML Class Diagram, and the control flows using the UML Activity Diagram.

Our Visualization Framework further helps the policy writer to statically analyze the security policy written, with respect to the possible execution sequences in the underlying bytecode. Our approach identifies the possible execution sequences in the program and maps each execution sequence to the set of corresponding state transitions of the security automaton (given by the security policy). A user-defined “color code” is provided to visually map the control flow blocks to the corresponding security automaton states. The resulting control flow diagram is a collection of color coded control flows of the program that identify the possible security violations and security-relevant program operations.

We take a conservative approach in which the detection of policy violations may include some false positives. For example, a potential violation might be identified within an execution branch that may not be traversed at runtime based on the value of some input variable. However, false negatives will not occur; all possible security violations are detected. To validate our tool we have written a test application that causes an information leakage over the network and a security policy that prohibits any send operation after a read has been executed. On applying the security policy to the test program, we manually validated the output visual model with the expected results (detection of policy violations).

A prototype tool has been developed as a proof of concept. A test client server application has been written, with a prototype security policy to avoid information leakage on the network, to test and validate security aware diagram specifications generated by the tool.

1.5 Organization of thesis

The rest of the thesis is organized as follows: Chapter 2 discusses the existing tool support for analysis of bytecode and the related work for software visualization. Chapter 3 outlines the visualization framework proposed for security analysis. Chapter 4 describes the validation for the prototype tool developed. Chapter 5 discusses the future work for the visualization for security analysis.

CHAPTER 2

RELATED WORK

2.1 Introduction

Practical IRM systems constitute a growing body of past work (c.f., [2]). These systems employ enforcement mechanism for given specification of a security policy [5, 6]. The formulation and testing of these prototype, security policies however remains a largely manual task. This section discusses the existing tool support that aids in the formulation of security policies. The available tools follow a textual or a graphical approach. We examine each of these individually and then summarize the approach proposed by this thesis.

2.2 Code Level Visualization: Textual

Traditional text-based, code-level visualization is supported by established tools including decompilers, debuggers; libraries are available for static code analysis [9, 10] [15]. These tools are general purpose; they do not provide specialized support for security. For example, code related to a specific security policy is not identified by highlighting the code.

Several decompilers are available for the Java bytecode. Amongst the popular ones are JAD - the fast JAva Decompiler[9], is available free for non-commercial use. Various GUI based front-ends are available for JAD, including DJ Java Decompiler, Cavaj and JadClipse. Another useful commercial decompiler available is the SourceAgain decompiler [10]. A decompiler gives an estimate of the original source code, generated from its low-level

executable code. This source code can then be examined manually to understand the application structure and to identify the possible control flows.

Debuggers for source code [11] and bytecode are available. Eclipse provides an integrated debugger for the java source code that allows executing the source code interactively, by stepping through each line of code. Some open source byte code debuggers are also available for finding the trace of execution at binary code level.

The BCEL API (Byte Code Engineering Library) [15] also provides a set of APIs for the static analysis of Java bytecode. These APIs can be used to print out all the necessary information about the bytecode structure. However, it still remains a text-based analysis to determine the application structure and possible control flows from the low-level bytecode.

Table 2.1 given below, summarizes the text-based approach for bytecode analysis.

Table 2.1. Text-based approach for bytecode analysis

	Decompilers/ disassembled	Debuggers	Static analysis libraries, BCEL
Code	Bytecode	Source code/ bytecode	Bytecode
Visual modeling notation	None	None	None
Static/ dynamic support	Static textual information	Dynamic (runtime) analysis of bytecode	Static textual information
Explicit support for security	None	None	None

2.3 Code Level Visualization: Graphical

As the size of the application increases, it becomes quite cumbersome to do a text-based analysis of the code. In addition, some execution traces may be overlooked or missed during the manual code analysis. A graphical model of the low-level code provides a tool for easy, faster and more reliable analysis of the application structure and the possible control flows. The main quality attributes and functional requirements for visualization tools have been identified; however, security is not specifically considered [16]. The suggested quality attributes require the visualizer's rendering speed to scale to larger amounts of data. To avoid an information overload to user on the interface, detailed information should be hidden in the main view but easily available to the user on a mouse click. A visualization tool is desired to be interoperable with other tools, customizable, interactive and explorative in nature, easy to use and easy to adopt by the user's environment. The functional requirements specify having different views of the target system, providing different levels of abstraction, and having search and filter features. It further requires providing a fast and easy access to underlying source code, automatic layout capabilities and the possibility to record the history and to undo user actions.

Modeling and development tools are available that provide the capabilities to reverse engineer code to UML class and sequence diagrams [12]. These tools are also general purpose; they do not provide specialized support for security. For example, code related to a specific security policy is not identified by highlighting the related classes on a class diagram.

Software visualization has been very useful for research tools that provide reverse engineering of applications. Rigi, an environment for software reverse engineering, exploration, visualization, and re-documentation has been developed [13]. It provides a

visualization engine that integrates both interactive and automated functionality for reverse engineering. It uses a methodology of structural re-documentation that starts out by laying out a flat graph of lowest level artifacts and then iteratively builds upon these by method of grouping and refining until the desired level of abstraction is reached.

Context independent analysis [14] is another example of a visualization tool that has been developed for binary files whose underlying format is not known. The tool incorporates the functionality provided by a hex editor and enhances it using byte plot visualization for the underlying file. The tool gives visual cues to identifying byte presence, repeated sequences of bytes or regions of compression or encryption.

Table 2.2 given below, summarizes the available graphical tool support for bytecode analysis.

Table 2.2. Graphical approach to bytecode analysis

	Commercial Modelling tools - IBM software architect	Research Reverse engineering tools - Rigi	Binary Visualization tools - context independent analysis
Code	Source code/ bytecode	Bytecode	Binary code (unknown format)
Visual modeling notation	UML	Graph based	Byte plot visualization
Static/ dynamic support	Static and dynamic views supported as UML specifications	Static	Static analysis
Explicit support for security	None	None	None

2.4 Conclusion

These available commercial and research tools are proficient but leave most of the burden of performing a rigorous security analysis to the user. The modeling tools like the IBM software architect use UML as the modeling notation, which is a powerful and established standard for the graphical modeling of object oriented software design and analysis. The reverse engineering tools developed provide support for building abstraction over the low-level code. However, these tools do not support for more specific, security related tasks like identifying the control flows in the application, mapping candidate policies on the control flow and testing prototype policies for identifying security violations. We take the approach of catering to these security-related requirements, while utilizing the powerful features of a visual modeling notation like UML and using the concept of abstraction for a better representation of the underlying bytecode.

The BCEL API (Byte Code Engineering Library) [15] provides a programmatic foundation for analyzing Java bytecode. Our visualization framework uses this API to extract low-level Java bytecode information.

Graphical models of low-level code provide easier, faster, and more reliable analysis of an application's structure and its possible control-flows than their text-only counterparts. We therefore adopt a graphical approach. Desirable quality attributes and functional requirements for general-purpose code visualization tools have been well-studied [16], but there has been no similar study of security-aware tool functionality to our knowledge.

CHAPTER 3

VISUALIZATION FRAMEWORK FOR SECURITY ANALYSIS

3.1 Introduction

The visualization framework provides a graphical model for easier, faster, and more reliable analysis of an application's structure and its possible control-flows. The visual diagram model is represented as the UML notation. We use UML diagrams to represent the structure of the underlying bytecode to aid the required security analysis. We hence emphasize on identifying and visualizing the selective UML constructs that are required for the security analysis. The implementation of the entire UML specification is left for future work. This section describes the proposed architecture for the visualization framework and the approach taken for generation of the UML-based, security-aware diagram views. A prototype tool has been developed as a proof of concept and used for the validation. A hand-written client server application, that causes information leakage on the network has been used as an example throughout the chapter. (see Appendix A for the example application code)

3.2 Overview

The Visualization Framework transforms the low-level bytecode into UML-based visual models. The framework is composed of a controller and separate view engines for the generation of static and dynamic visual models (see Figure 3-1). Based on the user request, the controller delegates control to the respective view engine. The static views engine

generates the view for the static structure of the application modeled as the UML Class Diagram. The dynamic views engine generates the dynamic views for the detailed control flow diagrams for each class method within the application code, modeled as the UML Activity Diagram.

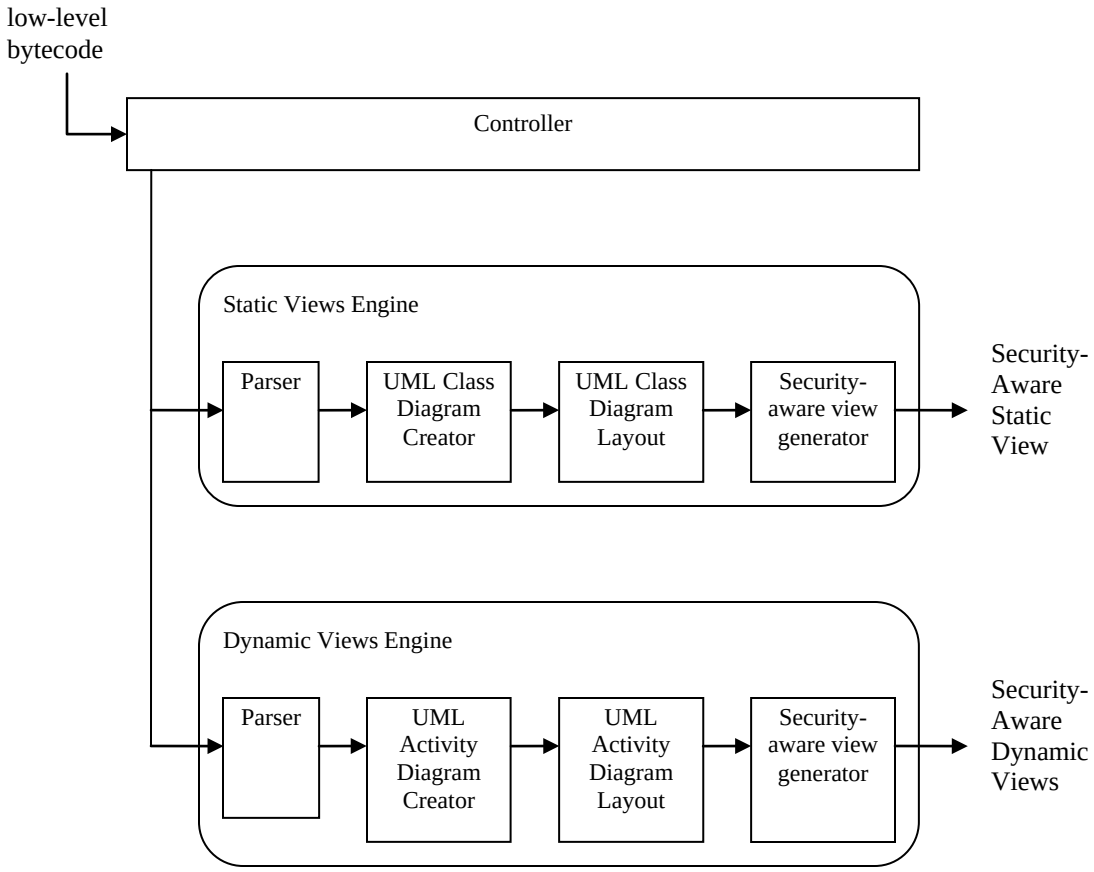


Figure 3.1. Security-Aware Visualization Framework Overview

These UML specifications are used to provide the security-aware views of the application. The UML Class diagrams can be used to compare the untrusted bytecode with the self-monitoring, rewritten code obtained by enforcing the security policy. The UML

Activity diagrams can be used to map the prototype security policy to the underlying control structure and to visually identify the security events in the control flows. The UML Diagrams are generated using an incremental graph generation approach. This approach taken for the security-aware static and dynamic view generation is described in more detail in Sections 3.2 and 3.3.

3.3 Security-Aware Static View

To effectively prototype and analyze real IRM's and the policies they enforce, it is important to be able to easily visualize and compare the class structure of original and IRM-modified Java bytecode applications. For example, most practical policies constrain usage of certain security-relevant system classes by untrusted applications. The IRM must therefore track the security state of these security-relevant objects at runtime to enforce the policy. The IRM typically accomplishes this by injecting wrapper classes that inherit from and extend the system classes with extra security state fields maintained by the IRM [5, 6]. Thus, visualizing the class structure of original and IRM-modified applications reveals much about the potential effects of the policy-enforcement upon the untrusted application, including undesired side-effects and potential runtime overhead.

The static class structures of applications are modeled as the UML Class Diagrams [8]. They define the structure of classes and relationships between them. A class diagram can be represented as a graph with nodes as the UML class elements and the edges as the relationships between them.

3.3.1 Graph model representation

The security-aware UML Class Diagram is represented as a graph G=(N,E), where N is the set of nodes in the graph and E is the set of edges in the graph. The set of nodes N denote the class elements of a UML Class Diagram. Each node has an associated tuple to represent the various attributes and methods associated with the class. For each node N, we have the tuple definition N=(name, access, type, attributes, operations) where,

name: the string attribute denoting name of the class

access: the string attribute denoting the access modifier(visibility attribute) for the class

type: the string attribute used to identify a class added by the security mechanism

attributes: the data attributes contained by the class

operations: the operations contained by the class

The set of edges E denote the relationships between the classes in a UML Class Diagram. Each edge has an associated tuple defined as E=($origin_N$, $target_N$) where, $origin_N$ represents the class element where the relationship originates and $target_N$ represents the target class element of the relationship. The set of edges E can be further divided into two sets $generalization_E$ and $association_E$, to denote the UML generalization and association relationships. These edges are defined as below,

A generalization edge $generalization_E$ is defined as the subclass-superclass relationship between classes, $generalization_E$=($origin_N$, $target_N$) where

$origin_N$: represents the class element that is the subclass

$target_N$: represents the class element that is the superclass of $origin_N$

An association edge $association_E$ is defined as the reference relation between the two classes, $association_E$=($origin_N$, $target_N$) where

$origin_N$: represents the class element that contains a call reference to $target_N$

$target_N$: represents the target class element whose reference is called by $origin_N$

3.3.2 Incremental graph generation approach

The graph model representing the security-aware static view (UML Class Diagram) is incrementally generated over the four pipelined components of the Security-Aware Static Views Engine– the Bytecode Parser, UML Class Diagram Creator, UML Class Diagram Layout, Security-Aware View Generator. Figure 3.2 summarizes this incremental graph generation approach.

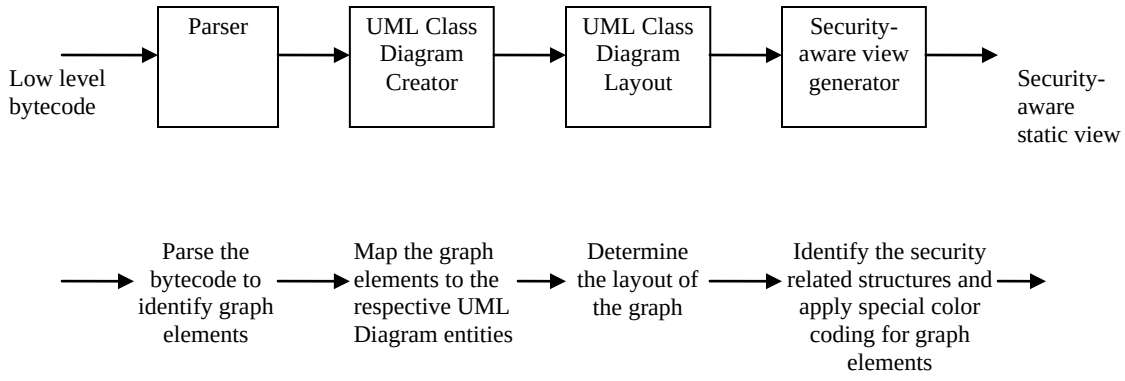


Figure 3.2. Pipelined components of Static Views Engine

The Bytecode Parser identifies the graph elements from the underlying bytecode. The graph nodes N represent the classes, their data attributes and methods. The visibility options of the data attributes and methods are extracted. The relationships between the classes are also parsed from the bytecode and are represented as the edges E of the graph. The UML relationships supported include generalization and association which are represented in the

graph using edges $generalization_E$ and $association_E$ respectively. The generalization relationship is identified as the inheritance relation between the superclass and its inheriting subclass. The association relationship is identified as references to class objects [7]. The table 3.1 defines the approach taken by the bytecode parser to identify the graph elements.

Table 3.1. Algorithm to identify graph elements

readGraphElements:
    forall ".class" files in the jar
        N ← instantiate class parser to read the classes in this file
    end **forall**

    forall clazz ∈ N
        $generalization_E$ ← (clazz, superclass of clazz)
    end **forall**

    forall instr ∈ InstructionList contained by each clazz ∈ N
        if instr instanceof InvokeInstruction
          $association_E$ ← (invoking class, invoked class)
        end **if**
    end **forall**
end **readGraphElements**

Note: The class parser to read .class files, extracting superclass of each clazz, the instruction-list's to work on and identification of the invoking class and invoked class is done using the BCEL api.

For the test client server application, the bytecode parser extracts all the information about the classes in the application, their data attributes, operations, superclasses and the relationships between the classes identified. For example, the main function of the Client class contains the bytecode:

```
0:  new             <ClientSocket> (16)
3:  dup
4:  invokespecial   ClientSocket.<init> ()V (18)
```

```
7:   astore_1
8:   aload_1
9:   ldc              "Test" (19)
11:  bipush                 7
13:  invokevirtual   ClientSocket.createSocket (Ljava/lang/String;I)V (21) .....
```

Since the Client class creates an instance of the ClientSocket class (at byteoffset 0) and then further invokes the methods of the ClientSocket class (at byteoffset 4, 13), the bytecode parser identifies this as an association relation between the two classes.

The Diagram Creator generates the entities of the UML Class Diagram that represent the graph elements extracted from the bytecode by the parser. The classes in the bytecode are mapped to the UML Class Elements and the relationships between these classes are mapped as the generalization or association relationships of the UML Class Diagram. The UML metamodel identified therefore consists of the ClassElements, data attributes, operations, visibility attributes, generalization and association relationships. Table 3.2 illustrates the mapping method used to create the UML Class Diagram entities.

We construct the UML Class Diagram with inheritance relationships up to one level into the system libraries. Since every class in Java inherits from the java.lang.Object by default, we have a strong inheritance based structure for the class elements.

For the test application developed, the UML Diagram generated hence includes the system library class java.lang.Object alongwith the application classes Server, Client, ClientSocket. The generalization relationships are identified between the java.lang.Object class and each of the application classes, since they inherit this class by default in Java. The association relationship identified from the bytecode is between the Client and ClientSocket class. These UML Diagram entities identified are then input as graph elements to the Layout algorithm, to generate the visual diagram.

Table 3.2. Mapping graph elements to UML Class Diagram entities

maptoUML: (N,E):ClassDiagramEntities
    forall n ∈ N
        create a new UML class element
        $classElement_{UML}$ ← n
        Set the details for the class: jvm class/interface
        $classElement_{UML}$ ← setDetails
        Set the data attributes for the class
        $classElement_{UML}$ ← setAttributes
        Set the methods contained by the class
        $classElement_{UML}$ ← setOperations
        ClassDiagramEntities ← $classElement_{UML}$
    end **forall**

    forall g ∈ $generalization_{E}$
        create a new UML generalization
        $generalization_{UML}$ ← g
        ClassDiagramEntities ← $generalization_{UML}$
    end **forall**

    forall a ∈ $association_{E}$
        create a new UML association
        $association_{UML}$ ← a
        ClassDiagramEntities ← $association_{UML}$
    end **forall**
end **maptoUML**

The UML Diagram Layout takes an approach of using the inheritance based structure for the automatic layout generation of the Class Diagram. Graph drawing algorithms have been used for generating automatic layouts of software diagrams. An inheritance based approach has been utilized in [17] for UML Class Diagrams defined for architectures that have a considerable use of the inheritance/generalization relationships. Since our diagrams have a strong inheritance based structure, we use a simplification of the algorithm.

The Class diagram represented as a graph model is input to the layout algorithm. The algorithm decides the layout of the graph by positioning the nodes on horizontal layers L.

Each horizontal layer Li consists of a set of nodes Ni. The following summarizes the three-step adaptation of the algorithm in [17] for automatic layout generation of the class diagram with minimal edge cross-overs. (see table 3.3 for the detailed algorithm)

1) Generalization-based layering: Start with the nodes that represent JVM class elements placed on layer L_0. Iteratively position the remaining nodes such that nodes that inherit from layer L_i are positioned on layer L_{i+1}.

2) Association-based reordering: To minimize the edges crossing over class elements, classes having an association relationship should preferably be placed adjacently. For each association, position the two nodes as neighbors if possible. If this is not possible, move both nodes to a newly inserted new layer immediately below the current one.

3) Offset calculation: To position each class within its layer, compute a base-point for each class, where a basepoint is defined as the minimum offset required for all its subclasses relative to the position of the superclass. The class' final offset is computed as the sum of the offsets of the nodes preceding this class on the current layer, plus its base-point.

This algorithm generates a simple layout with minimum edge cross-overs. The visualizer further allows the users to manually adjust the generated layout by providing a select, drag and drop functionality for the class elements of the diagram.

Screen-shot 2 shows a UML Class Diagram generated using the framework, for the test client-server application. The classes are positioned on layers as per their inheritance graph structure (step 1), level 0 contains the system class java.lang.Object and the application classes form the level 1. The Association based re-ordering, (step 2) decides the order of classes on each level, at level 1, the classes Client and ClientSocket are made neighbors since they have an association relationship. Finally the x-axis offsets for each layer are computed

(step 3), which completes the positioning of the classes by computing their x-axis co-ordinates on each layer.

Table 3.3. Layout algorithm for the UML Class Diagram

Layout: (N, L)

Step 1: Generalization-based layering:

forall $n \in N \wedge n \in$ jvm class elements,

$L_0 \leftarrow n$

$N \leftarrow N - n$

end **forall**

forall $l_1, l_2 \in L_0$,

if l_2 inherits from l_1 then $L_1 \leftarrow l_2$

end **forall**

forall $n_i \in L_i$

if $n \in N$ inherits from n_i then

$L_{i+1} \leftarrow n$

$N \leftarrow N - n$

end **if**

end **forall**

Step 2: Association-based reordering:

compute the set of association edges $I=(origin_N, target_N)$, $I \subseteq E$

if $origin_N, target_N \in L_i$

Position the $origin_N$ and $target_N$ to be neighbors if possible,

else insert a new layer L_{i+1} and $L_{i+1} \leftarrow target_N \cup origin_N$

Step 3: Offset calculation:

forall $n_i \in L_i$

$basepoint_n \leftarrow$ minimum offset based on position of parent class

relative_offset $\leftarrow basepoint_n$ + offsets introduced by the nodes positioned before n_i on Li

end **forall**

end **Layout**

The framework can additionally render a visual comparison between the original and the rewritten, IRM-modified application bytecode. The Security-Aware View Generator decides on the color coding for the graph (diagram) elements. It compares the original and the rewritten bytecode, over their class structures and the new classes introduced by the IRM are visually highlighted, (see Table 3.4). This is extremely useful for analyzing changes in the static structure that result from enforcement of a given policy by an IRM. Separate UML class diagrams are generated for the original and IRM-modified application bytecode, with visually highlighted color cues for the security-relevant classes introduced by the rewriter.

Table 3.4. Compare algorithm to highlight the classes added by the IRM

```
Compare: (N_original, N_rewritten): N_new
        forall n ∈ N_rewritten
            if n ∉ N_original
               N_new ← n
            end if
        end forall

        forall n ∈ N_new
            set a highlight for the fillcolor of n
        end forall
end Compare
```

Screen-shot 1 demonstrates the UML Class Diagram views generated for a test application that opens a file and then creates a socket connection. We enforce a security policy that prohibits a call to the socket class once a confidential file has been opened for read. The IRM introduces a security class during the process of re-writing. This new security class added by the re-writer is highlighted in the generated diagram view for comparing the

two bytecode applications. The next section discusses the detailed generation of the security-aware static view using the framework.

3.3.3 Example for security-aware UML Class Diagram

To illustrate the generation of the security-aware static view, we use a hand-written client-server application that opens a file from a local directory and then proceeds to establish network connections. We used the SPoX system to enforce a security policy on the bytecode of this application. The security property defined is that no network connection can be allowed once a local file has been opened for access. The visualizer was used to generate a comparison of the class structures for the original and the rewritten bytecode. The two separate class diagrams generated by the tool are shown in screenshot 1. The following is a walk-through for the approach taken by the tool to generate the security-aware UML Class Diagrams for the test application.

The bytecode parser extracts the class structure information from the bytecode using the BCEL API. The static components of the bytecode are identified using the `org.apache.bcel.classfile` package of the BCEL library. The top level data structure in BCEL is the JavaClass that corresponds to the class in the underlying bytecode.

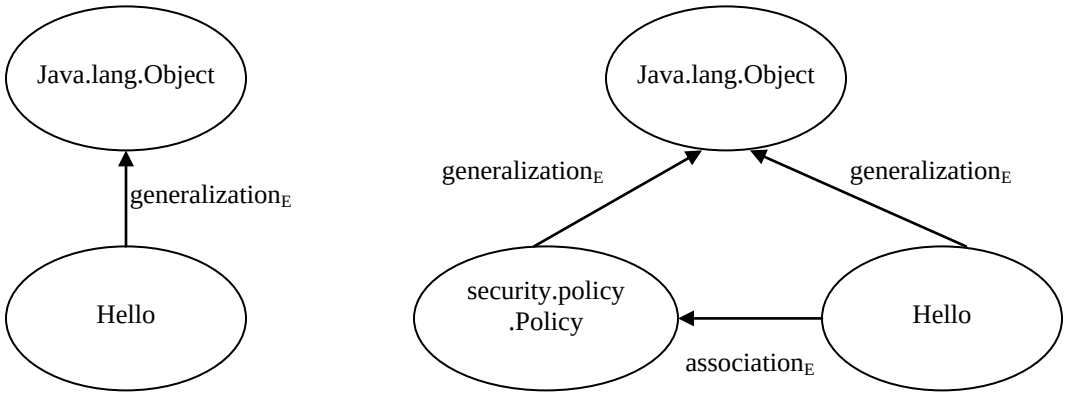
The JavaClass components for the original bytecode are identified as the classes java.lang.Object, Hello. The subclass-superclass relationship between class Hello and class java.lang.Object is obtained from the JavaClass api. The graph elements identified for the original bytecode are illustrated in table 3.5.

A similar approach is taken for the re-written bytecode. The bytecode parser identifies the JavaClass components corresponding to the classes java.lang.Object, Hello,

security.policy.Policy. The subclass-superclass relationships are identified between (class Hello and class java.lang.Object), (class security.poilcy.Policy and java.lang.Object). The class reference to security.policy.Policy class is found in bytecode of class Hello as shown below.

```
// Method signature: ([Ljava/lang/String;)V
main();
Code(max_stack = 3, max_locals = 4, code_length = 48)
......
7:    invokestatic     security.policy.Policy.edge_hasRead ()V (46)
......
```

These components identified from the bytecode are translated as the graph elements for the rewritten bytecode. The classes identify the nodes N of the graph, the subclass-superclass relationships identify the generalization edges $generalization_E$ of the graph and the class references identify the association edges $association_E$ of the graph. The underlying graphs generated for the original and rewritten bytecode are shown in figure 3.3.



Graph generated for original application bytecode

Graph generated for rewritten, IRM-modified application bytecode

Figure 3.3. Graphs generated for the original and rewritten application bytecode

The UML Class Diagram generator generates the UML Class Diagram entities corresponding to these graph elements. The nodes N form the class elements, edges $generalization_E$ form the generalization relationships and edges $association_E$ form the association relationships of a UML Class Diagram.

The UML Class Diagram Layout component then generates two separate class diagram layouts for the original and rewritten bytecode views, using the graph-based algorithm explained in section 3.3.2.

The security-aware view generator compares the graph elements of the original and rewritten bytecode to detect the new classes added by the IRM system, (see table 3.5). As we can see in this example the IRM has introduced a new security.policy.Policy class to the application. The security-aware view generator then adds visual high-lighting to the generated UML Class Diagram and the security-related class security.policy.Policy is highlighted in its corresponding Class Diagram.

Table 3.5. Comparison of graph elements for the original and rewritten bytecode

Graph elements for original bytecode	**Graph elements for rewritten bytecode**
N={java.lang.Object, Hello}	N={java.lang.Object, Hello, security.policy.Policy}
$generalization_E$={(Hello, java.lang.Object)}	$generalization_E$={(Hello, java.lang.Object), (security.policy.Policy, java.lang.Object)}
$association_E$={}	$association_E$={(Hello, security.policy.Policy)}

3.4 Security-Aware Dynamic View

To identify the possible security violations defined by history-based security policies, the sequences of security-relevant events occurring in the execution need to be tracked. This involves identifying all possible control flows in the execution. A visual model for the control flows in the application becomes important for examining every possible flow of execution in the application, in a faster and more reliable manner. Further the policy writer needs to analyze if a particular control flow will reach a security violation, for a given candidate security policy. The visualization for the effect of applying this security policy to the application (resulting in detection of possible security violations) aids in rapid formulation and testing of prototype security policies.

The control flows in an application are illustrated using the UML Activity Diagrams [7, 8]. Activities typically consist of a graph of nodes and edges, that represent the flow within the activity.

3.4.1 Graph model representation

The UML Activity Diagram can be denoted as a graph G=(N,E), where N is the set of nodes in the graph and E is the set of edges in the graph. The set of nodes N denote the nodes in a UML Activity diagram. The set of nodes is divided further to represent the supported UML activity nodes– the set of nodes callaction $_N$ denote the call action nodes and can be defined as callaction$_N$=(basicblock) where, basicblock is the contained instruction list that forms a basic block(as explained in section 3.4.2). The control nodes–initial node, exit node and decision node of the UML activity diagram are represented by initialnode$_N$, exitnode$_N$ and decisionnode$_N$ respectively. The decision node set can be defined as

decisionnode$_N$=(targetList) where, targetList defines the list of decision targets contained by the node.

The control flow edges of a UML Activity diagram are denoted by the set of edges E, defined as E={(initialnode$_N$, callaction$_N$), (callaction$_N$, callaction$_N$), (callaction$_N$, decisionnode$_N$), (decisionnode$_N$, callaction$_N$), (callaction$_N$, exitnode$_N$)}. These define all the control flow edges in the activity diagram, including the exception handler edges that have a special visual representation (as explained in section 3.4.2).

3.4.2 Incremental graph generation approach

The graph model representing the security-aware dynamic view (the UML Activity Diagram) is incrementally generated over the four pipelined components of the Security-Aware Dynamic Views Engine– the Bytecode Parser, UML Activity Diagram Creator, UML Activity Diagram Layout, Security-Aware View Generator. Figure 3.4 summarizes this incremental graph generation approach.

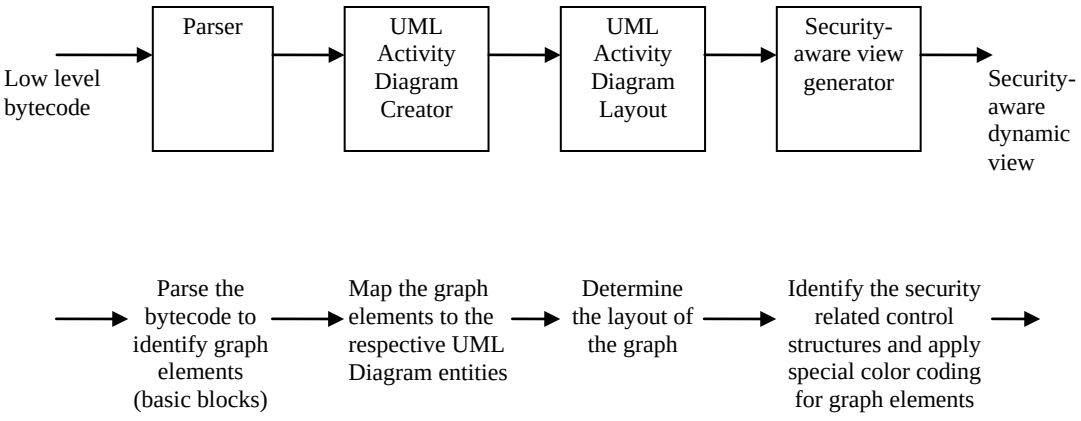


Figure 3.4. Pipelined components of Dynamic Views Engine

The Bytecode Parser parses the sets of instructions from the bytecode that form the basic blocks. A basic block is a sequence of consecutive bytecode instructions for which the control flow enters at the beginning and leaves at the end without halt or branching [18] (other than exceptions, which receive special treatment described below). Basic blocks have one entry-point and one exit-point. A basic block entry-point is identified as:

- The first instruction of a method.
- Each instruction that is target of an unconditional or conditional branch instruction. (All branch targets are static in Java bytecode.)
- Each instruction that immediately follows a branch instruction.
- Each instruction that is the start of an exception handler.

Each entry-point identifies a basic block in the system and each basic block includes instructions starting from the entry-point up to and not including the next entry-point in sequence.

The bytecode parser identifies the entry-points and the basic blocks in the underlying bytecode. Table 3.6 demonstrates the detailed algorithm for identifying the basic blocks from the bytecode of a method of a given class.

The identified basic blocks are used to generate the complete graph for the dynamic view. The basic blocks form the nodes and the control flow structures (branch instructions of the basic blocks determine the control flow edges) Table 3.7 illustrates the algorithm to identify all the graph elements.

Table 3.6. Algorithm to identify the basicblocks from the bytecode

```
IdentifyBasicBlocks (InstructionList):basicBlocks_N
        forall instr ∈ InstructionList
            if instr is first instruction
               entrypoint ← bytecode offset for instr
            end if

            if instr is branch instruction
               entrypoint ← bytecode offsets of all targets of instr
               entrypoint ← bytecode offset of instr immediately
                            following this branch instr
            end if

            if instr is start of an exception handler
               entrypoint ← bytecode offset for instr
            end if
        end forall

        sort the recorded entrypoint and eliminate duplicate entries

        forall instr ∈ InstructionList
            if bytecode offset for instr ∈ entrypoint
              create new basicblock basicblock_i+1
              basicblock_i+1 ← instr
              record the previous complete basicblock to output
              basicBlocks_N ← basicblock_i
            else
              add to the previous basicblock
              basicblock_i ← instr
            end if
        end forall
end IdentifyBasicBlocks

Note: The bytecode offsets are obtained using the BCEL api's.
```

Table 3.7. Algorithm to identify the graph elements

```
readGraphElements:
        create the initial node
        N ← initialnode_N
        BasicBlocks_N ← IdentifyBasicBlocks
        forall basicblock ∈ BasicBlocks_N
            callaction_N ← basicblock
            N ← callaction_N
            if basicblock contains a branch instruction
              decisionnode_N ← target bytecode offsets
              N ← decisionnode_N
            end if
        end forall
        N ← exitnode_N

        forall n ∈ N
            if n ∈ decisionnode_N
               record all the edges corresponding to the target bytecode offsets
               e ← (n, node containing target bytecode offset)
               E ← e
            else
               e ← (n, next node in sequence)
               E ← e
            end if

            if n contains an exception handler
               add the corresponding exception handler edge, this edge is
               marked as exception handler edge to draw it as a dashed edge
               e  ← (n, node containing the corresponding exception handler)
               E ← e
            end if
        end forall

end readGraphElements
```

The UML Activity Diagram Creator maps these basic blocks as the Call Action nodes of the activity diagrams, $callaction_N$=(basicblock). Call Action nodes define the units of work that are atomic within the activity [8]. Since the basic blocks define set of operations that are all executed sequentially without a halt or branch, we map these as Call Action nodes of the

UML Activity Diagram. Each basic block that ends in a conditional/compound conditional branch introduces the decision nodes in the diagram ($decisionnode_N$) and the target of the decision form the targetList for the $decisionnode_N$. Basic blocks that could throw exceptions caught by a local handler are visualized via control-flow edges from the basic block to the entry-point of the local exception handler. For visual clarity, the control-flow edges to exception handler blocks are shown as dashed arrows to differentiate from the normal control flow edges. Table 3.8 gives the map function used to generate the UML Activity Diagram entities.

Table 3.8. Algorithm to map graph elements to UML Activity Diagram entities

```
maptoUML: (N,E):ActivityDiagramEntities
        forall n ∈ N
            if n ∈ initialnode_N
               initialNode_UML ← n
               ActivityDiagramEntities ← initialNode_UML
            else if n ∈ callaction_N
               callActionNode_UML← n
               ActivityDiagramEntities ← callActionNode_UML
            else if n ∈ decisionnode_N
               decisionNode_UML← n
               ActivityDiagramEntities ← decisionNode_UML
            else if n ∈ exitnode_N
               exitNode_UML← n
               ActivityDiagramEntities ← exitNode_UML
            end if
        end forall

        forall e ∈ E
            create new control flow edge
            Edge_UML ← e
            if e is marked as an exception handler edge,
               Edge_UML ← mark as dashed edge
            end if
            ActivityDiagramEntities ← Edge_UML
        end forall
end maptoUML
```

The UML Activity Diagram constructs identified include the Call Action nodes, the control nodes that include Initial node, Final nodes, Decision nodes and the control flow edges. We use a dataflow analysis technique to identify all possible control flows in the control structure of the activity diagram by traversing them statically, (see Table 3.9). The UML Diagram Layout generates a flow-chart like layout that contains Call Action nodes ordered by the underlying bytecode offsets and the control flows between them.

Table 3.9. Dataflow analysis algorithm to statically identify all control flows

```
identifyControlFlows: (currentControlFlow, elementId)
        currentElement ← getDiagramElement(elementId)
        currentTop ← stack.peek()

        if currentElement is instanceof FinalNode
           record the currentControlFlow
           controlFlows ← currentControlFlow
        end if

        while true
              currentControlFlow ← currentElement

              if (currentElement contains a branch instruction ||
                currentElement has an exception handler)
                 break;
              end if

              if currentElement is instanceof FinalNode
                 record the currentControlFlow
                 controlFlows ← currentControlFlow
                 break;
              end if

              currentElement ← getDiagramElement(next elementId)
        end while
```

Table 3.9 continued

```
                if currentElement contains a branch instruction
                   add all its target bytecode offsets to the stack provided they
                   have not been traversed before and recorded in currentControlFlow
                   target ← elements containing target bytecode offsets
                   if target ∉ currentControlFlow
                      stack.push(target bytecode offsets)
                    end if
                end if

                if currentElement has an exception handler
                   add the exception handler block's offset to stack
                    stack.push(offset of exception handler)
                end if

                while stack.peek != currentTop
                        identifyControlFlows (currentControlFlow, stack.peek())
                        lastPopped ← stack.pop()
                        remove everything upto and including the lastpopped element
                        from the currentControlFlow
                end while

end identifyControlFlows
```

The Security-Aware View Generator further provides security-aware views of the activity diagram. It takes an input security policy from the user and maps the policy to the control flows depicted by the activity diagram. We detect all possible policy violations in the control flows using the algorithm described below.

Our visualizer computes a function $f:Q \rightarrow 2^S$ that maps each node $q \in Q$ in the control flow graph to a conservative approximation of the set of security automaton states $s \in S$ that

the node could assume during execution of the program. The computation involves obtaining the least fixed point of the functional F defined by

$$F(f) = f \cup \{(q_0,s_0)\} \cup \{(q',\delta(q,s)) \mid (q, q') \in E, s \in f(q)\}$$

where q_0 and s_0 are the start states of the control flow graph and security automaton (respectively), $E \subseteq 2^{Q \times Q}$ is the transition relation for the control flow graph, and $\delta:(Q \times S) \rightarrow S$ is the transition function for the security automaton, which defines how each basic block modifies the security state when executed. Our current dataflow analysis implementation is intra-procedural; an inter-procedural extension is left for future work.

The visualizer then identifies sites of potential policy violations by identifying the control-flow graph nodes $q \in Q$ for which there exists a security state $s \in (fix(F))(q)$ such that $(q,s) \notin \delta^{\leftarrow}$. These are the states for which the security automaton has no transition, and that therefore might exhibit a policy violation at runtime. These nodes q are therefore the sites where an IRM will typically implement runtime security checks to detect and prevent potential violations. The visualizer renders these nodes in a unique, user-specified color to bring them to the attention of the user.

On the implementation level, we maintain a hashmap to record the possible security states for each node q. We then iterate to discover the security states that the basic blocks may transition into. The algorithm terminates when the fixed point is reached—i.e., when no new security state transitions are detected for any node on a pass. Table 3.10 shows the pseudocode used for the implementation of the fixed-point algorithm.

The security states identified for each node are depicted using a color code provided by the user in the security policy. The color-coded control flows are provided to user on-

demand. A high-level color-coding for the control structure is included in the control flows that show all possible security states, allowing a view of all the possible security violations.

Table 3.10. Pseudocode for implementation of fixed point algorithm

```
FixedPoint: (Q, S):f
        while true
            track if an iteration over the nodes updates f
            node_state_changed ← false
            forall q ∈ Q
                compute all possible security states s S such that
                if (q, s)∉δ← ∧ (q, s)∉f
                    f ← (q, #)
                    node_state_changed ← true
                end if
                if (q, s) ∈ δ← ∧ (q, s)∉f
                    f ← (q, s)
                    node_state_changed ← true
                end if
            end forall
            if node_state_changed = false
                break;
            end if
        end while
end FixedPoint
```

We use a test client-server application that causes information leakage over the network to demonstrate the security-aware diagram generation. The security policy mapped on the generated control structure prohibits any send operation once a read has been performed. The visualizer generates the security-aware view of the control structure (see Screen-shot 4) that maps the possible security states of the automaton that each control flow block could enter, identifying the basic blocks where the security violation could result. The identified security states (including the security violation) is depicted using a color code

applied to the UML Activity Diagram. The generated highlevel view of the control structure (see Screen-shot 4) maps the possible security states of the automaton that each control flow block could enter, identifying the basic blocks where the security violation could result. In this case the yellow node indicates that the security automaton is in an initial state, the violet nodes indicate that the security automaton may be in various different policy-adherent states on various different runs, and the red node indicates a possible policy violation. The next section discusses in detail, the generation of this security-aware view.

3.4.3 Example for security-aware UML Activity Diagram

To illustrate the generation of the security-aware UML Activity Digram, we use a hand-written test client-server application whose control flow is simple enough to allow manual inspection. We focus on the main method of the Client, which is responsible for information leakage over the network. Further, we use a security policy that prohibits any send operations over the network, once a read has been performed. The security-aware view of the underlying control flows is generated by applying this security policy. We discuss in further detail the approach explained in section 3.4.2, using this test client server application.

The application bytecode is parsed by the bytecode parser component of the visualizer, to extract the basic blocks using the rules defined in section 3.4.2. Figure 3-5 shows the basic blocks identified for the main method of the client. These basic blocks form the callaction_N nodes of the graph. Each basic block that ends in a conditional branch instruction introduces a decisionnode_N to the graph. The initialnode_N and exitnode_N are identified as the start of the method and the return instruction. The set of control flow edges E is identified.

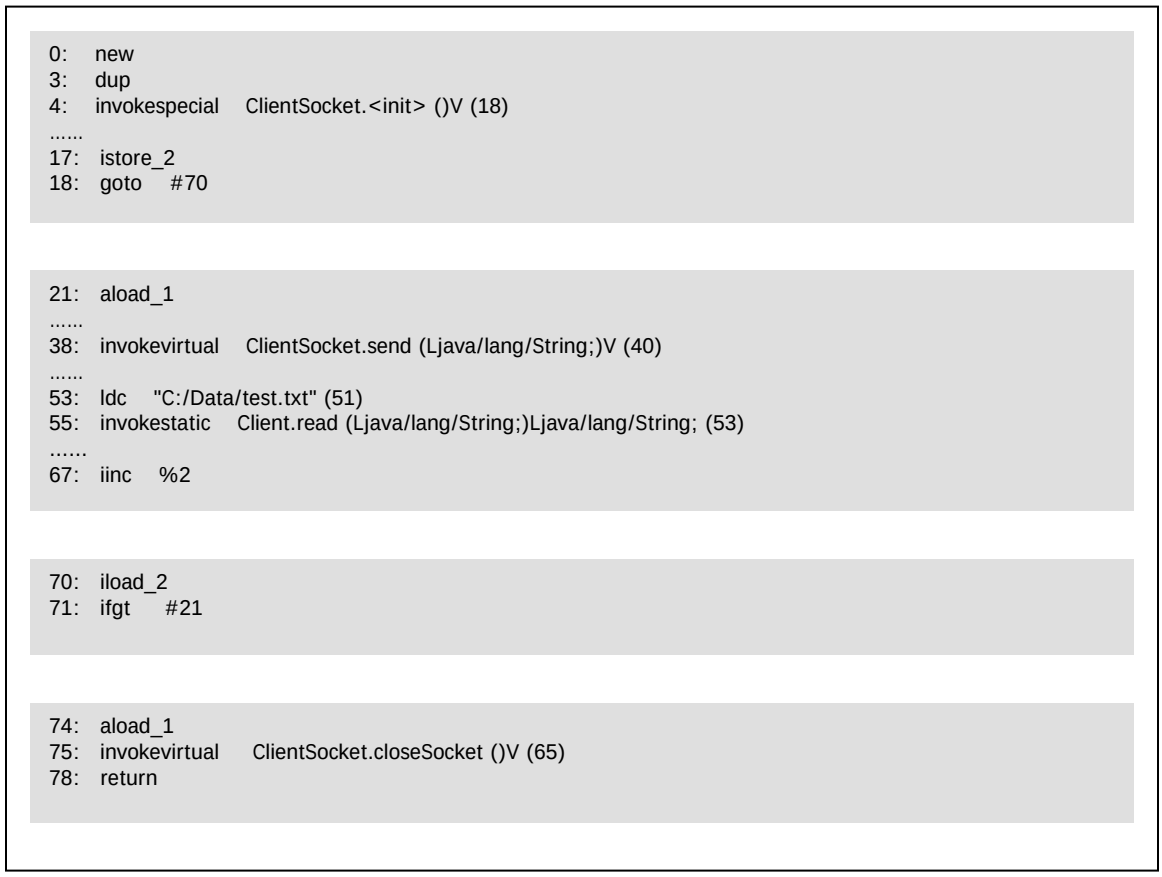


Figure 3.5. Basic blocks identified for Client.main method

The UML Diagram creator maps the graph elements to the UML Activity Diagram elements, $callaction_N$ are mapped to the call action nodes, the $initialnode_N$, $exitnode_N$ and $decisionnode_N$ are mapped to the control nodes of the activity diagram. The UML Diagram layout generates a flow-chart like layout based ordered by the underlying bytecode offsets. The security policy defined for avoiding the information leakage is then applied to the generated control flow. The main method of the client contains a loop, offsets 70, 71 form the loop condition and the loop body is from offset 21 through 67 (see Figure 4). We can see that the loop body includes a send operation (at offset 38) followed a read operation (at offset

55). The security policy mapped on the generated control structure prohibits any send operation once a read has been performed. This could result in a policy violation for control flows that execute two or more iterations of the loop body. The fixed-point algorithm computes the set of possible security states that could be reached by each of the basic blocks over the entire iterations of the loop. The function f computed by fixed point algorithm is,

f = {(basicblock$_1$, {0}), (basicblock$_2$, {0,1,#}), (basicblock$_3$, {0,1}), (basicblock$_4$, {0,1})}

As we can see the set of security states identified for basicblock2 includes the initial state 0, when the loop body is not executed even once, the state 1 relates to the scenario where the loop body is executed exactly once and finally the error state # is identified for the possible scenario of the loop body being executed more than once (resulting in a security violation).

A high level view of the control structure is generated by mapping the function f computed by the fixed point algorithm to the color codes defined in the security policy by the user (see Screen-shot 4). In this case we use the color coding, yellow to depict security state 0, red depicts the error state representing a possible security violation and violet indicates that the security automaton may be in various different policy-adherent states. This color-code applied to the generated UML Activity Diagram hence provides the security-aware dynamic view that identifies all possible security violations for the given policy.

3.5 Tool Support

The Visualization Framework is built on top of the Eclipse plug-in architecture. A plug-in is the smallest modular unit in Eclipse that contributes to functionality. Plug-ins can specify extension points that define the point where additional functionality can be extended by other plug-ins, [11]. Other plug-ins specify the extensions that implement this additional functionality. Figure 3.6 shows the plug-ins that compose the visualization framework. The

UDV is the main, high-level plug-in that controls the creation of various diagrams and navigation between them. This plug-in defines the extension points to add the functionality of generation and display of the diagrams. The level 2 plug-ins—ClassDiagram, ActivityDiagram and ByteCodeViewer—provide the extensions to generate the various diagrams and views.

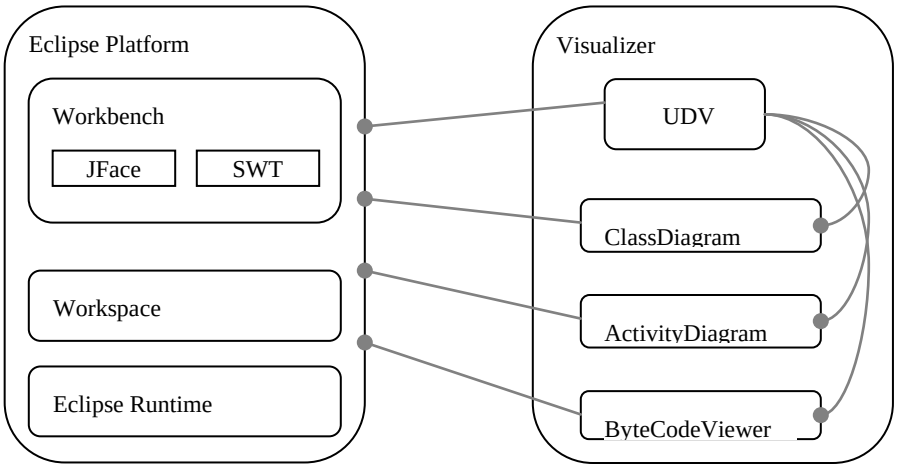


Figure 3.6. Platform Architecture

UDV plug-in: This plug-in is the Controller for the visualization tool. It is the main application that implements the IApplication interface and provides the main entry point to the tool. It uses the Eclipse platform runtime but controls the execution of the tool by itself. In addition it also includes a tree widget that allows the user to load the low-level bytecode, generate diagrams on-demand and navigate between the diagram views. This plug-in provides the extension points to extend the generation and display of the various diagrams.

The level-2 plug-ins extend the functionality of generating the diagram views. They form the view engines required for the generation of the visual models. These plug-ins

contain three main components: ByteCodeReader, Entities, and Views (Figure 3.7). The ByteCodeReader component provides the parser for the low-level bytecode; it parses the bytecode using the BCEL api's to extract the required information about the application structure. The Entities component encapsulates the graphical specifications for drawing the diagram entities as per the UML standard. The Views component provides the engine to integrate the data from the parser, identify the UML diagram entities, generate and layout the corresponding UML Diagram View.

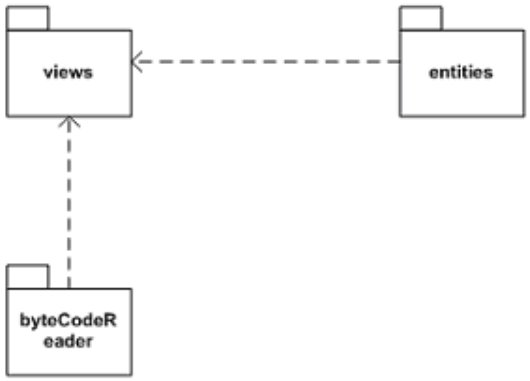


Figure 3.7. Conceptual components for diagram generation

ClassDiagram plug-in: This plug-in implements the Static Views Engine of the framework. It extends the functionality of generation of the diagram view for the UML Class Diagram using the automatic layout algorithm described earlier. The plug-in also provides the functionality to select, drag and drop diagram elements, to modify and refine the generated automatic layout manually. It further allows the reset or auto-adjust to original layout. In addition, the plug-in provides the functionality to compare the structures of two applications from their bytecode. This allows original and IRM-modified bytecode to be compared automatically, revealing how policy enforcement will tend to affect program structure. The

components of this plug-in are structured as shown in Figure 3.8. ClassDiagramVisualizer class forms the main views engine. ByteCodeReader class contains the bytecode parser. ToolTipHandler provides a custom SWT ToolTip implementation to provide a point-and-click mechanism to view underlying bytecode of the Class Elements. The Diagram class provides the container for all the diagram elements that constitute a UML Class Diagram.

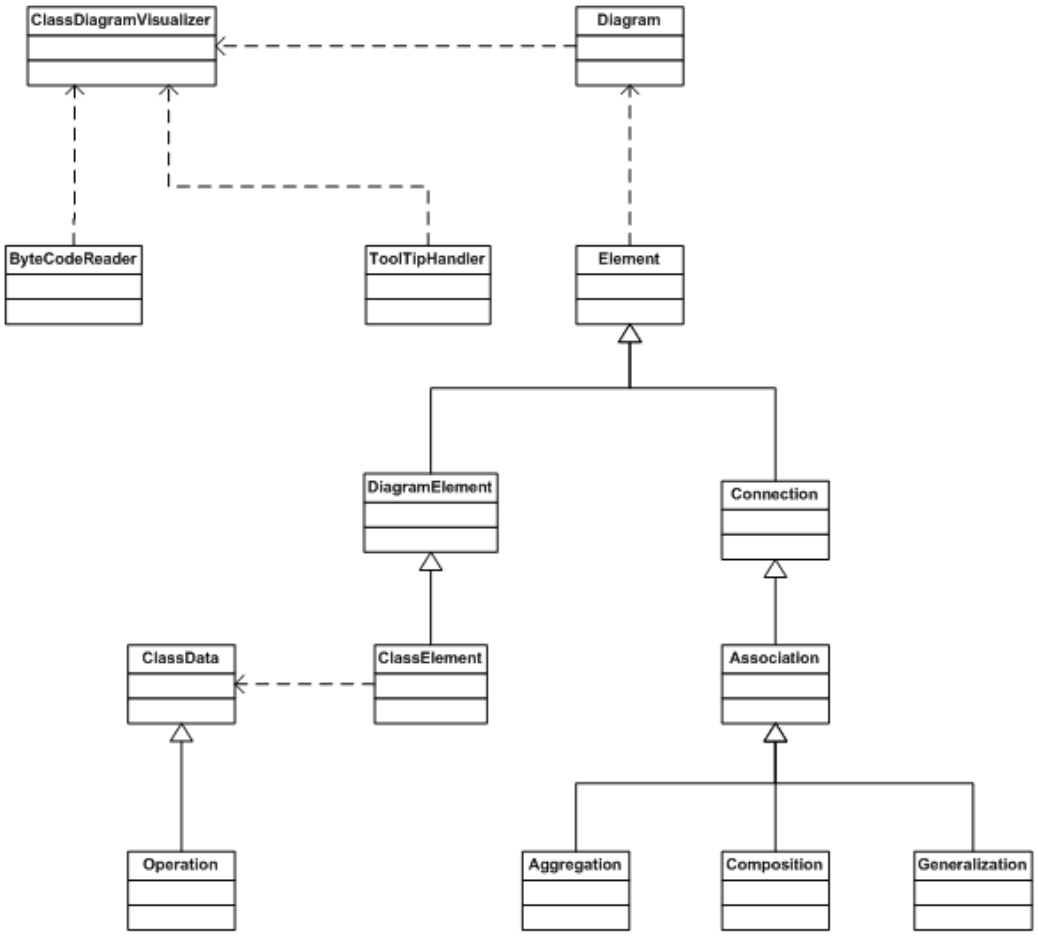


Figure 3.8. Structural view of the Class Diagram plug-in.

ActivityDiagram plug-in: This plug-in implements the Dynamic Views Engine of the framework. It extends the functionality of representing the control flow in a method as a

UML Activity Diagram. It further generates the security-ware dynamic views as described earlier, to validate the prototype policy written for the control flow by mapping the security automaton to the generated control structures. Each node in the diagram depicts a color-coded visual cue to the possible security state in the security automaton. The control flows are statically identified and visualized and all possible security violations are depicted. The plug-in is structured as shown in Figure 3.9.

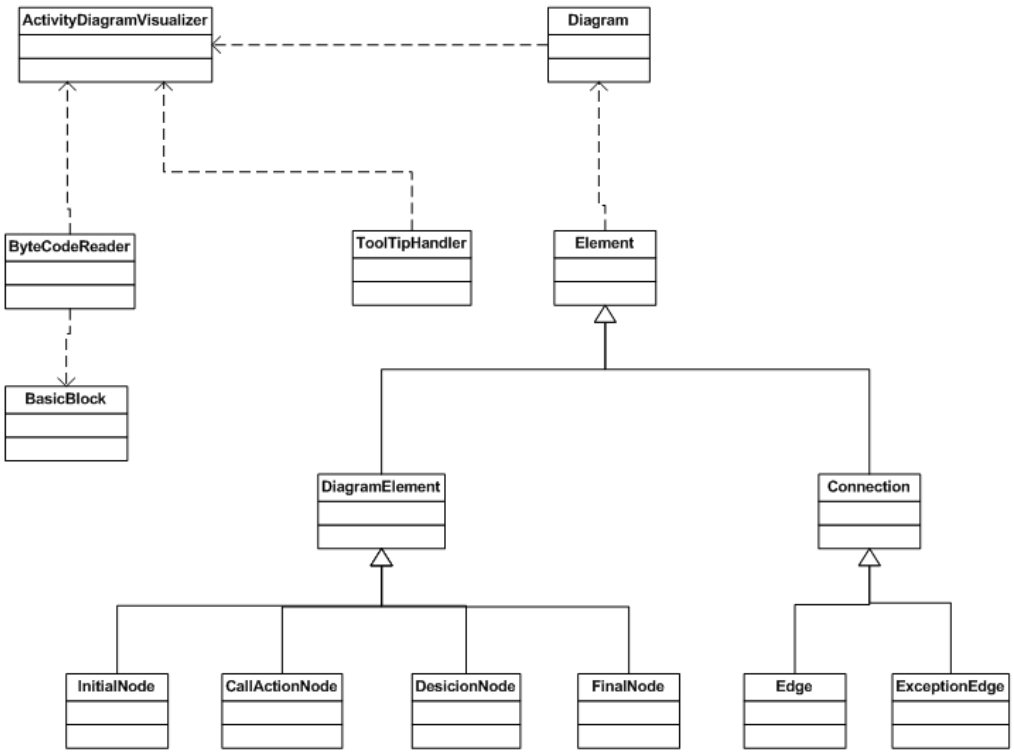


Figure 3.9. Structural view of the Activity Diagram plug-in.

The ActivityDiagramVisualizer composes the main views engine to create and layout the UML Activity Diagram Views. The ByteCodeReader class provides the bytecode parser that

identifies the BasicBlocks in the underlying code. A ToolTipHandler is provided to implement the point-and-click mechanism to view the bytecode instructions contained in each basic block. The Diagram class acts as the container for all diagram elements that constitute a UML Activity Diagram.

ByteCodeViewer plug-in: provides a view of the bytecode that is analyzed, at class-level and method-level. This provides a fast and easy access to the entire bytecode for the user's convenience.

Given the plug-in architecture, the visualization framework can be easily extended to add further custom diagrams related to the security specifications. It also provides for ease of adoption of the tool as plug-in perspective in the eclipse SDK or use as a standalone rich client desktop application. Further it does not require any special set-up, the only requirement being a java runtime environment.

3.6 Discussion

The visualization framework adheres to most of the quality attributes and functional requirements identified for software visualization [5]. A summary of the framework support for these has been demonstrated; see Table 3.11, Table 3.12.

Table 3.11. Quality Attributes for a software visualization tool

Quality Attribute	**Support in our framework**
Rendering scalability	The framework generates diagram views with small response time, and we found that the rendering speed scaled well to a medium-scale application with about 17 classes.

Table 3.11 continued

Information scalability	There is a limited amount of information displayed per view, and more information is accessible via point-and-click interaction.
Interoperability	As an Eclipse plug-in, the tool can be easily integrated with existing Java development environments or executed as a standalone application.
Customizability	Views can be customized via user-specified color-coding, drag-and-drop, and additional Eclipse plug-in extensions.
Interactivity	The tool is explorative in nature, showing different security-aware views on demand.
Usability	Provides an easy-to-use interface to users.
Adoptability	It is Eclipse-compatible and requires only a standard Java runtime environment.

Table 3.12. Functional Requirements for a software visualization tool

Functional Requirements	**Support in our tool**
Views	The framework provides separate views of the system to analyze the static structure of the underlying application and to view the control-flows of individual methods.

Table 3.12 continued

Code Proximity	The underlying bytecode of the application is available to the user at both class and method levels from all generated diagram views. This is achieved via a built-in bytecode viewer accessed by point-and-click. This provides for traceability and code proximity between the generated diagram view elements and their underlying bytecode.
Abstraction	An abstraction level is achieved by denoting the low-level bytecode as high-level elements of a control-flow in a class method or as class structures of the application. This is essential for quickly identifying pertinent code features and structures at a high level and then examining their details at a lower level.
Search	Could be a possible enhancement for the tool.
Filters	Could be a possible enhancement for the tool.
Automatic layouts	The tool uses a hierarchical layout for the class diagrams, along with ability to manually change the layout using a selection tool to drag and drop diagram elements on the diagram view.
Undo/History	Since the tool provides static analysis visualization, there is no edit operation for the user.

3.7 Screenshots

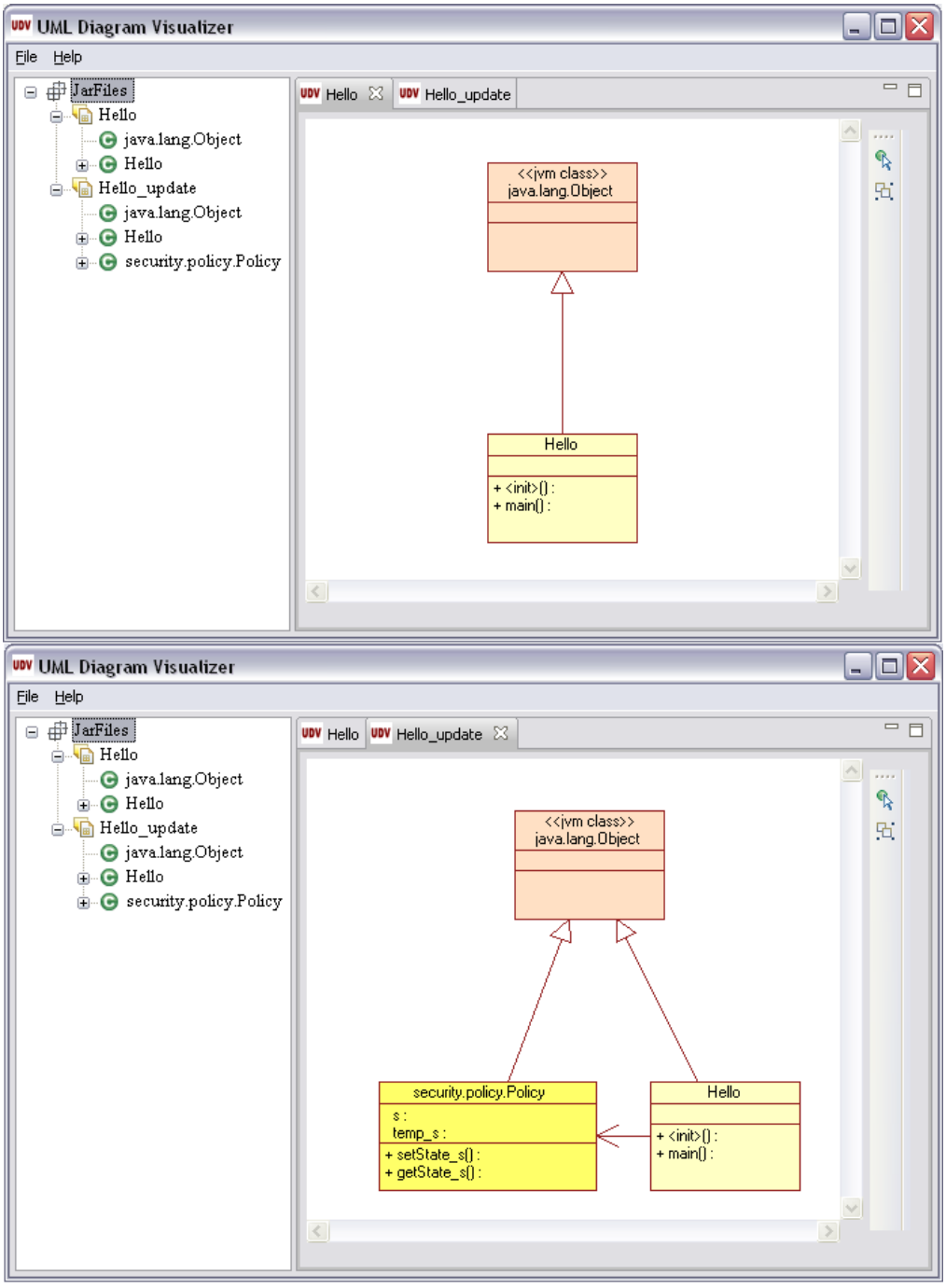


Figure 3.10. Screen-shot 1: Security aware UML Class Diagrams generated for the original and re-written application bytecode.

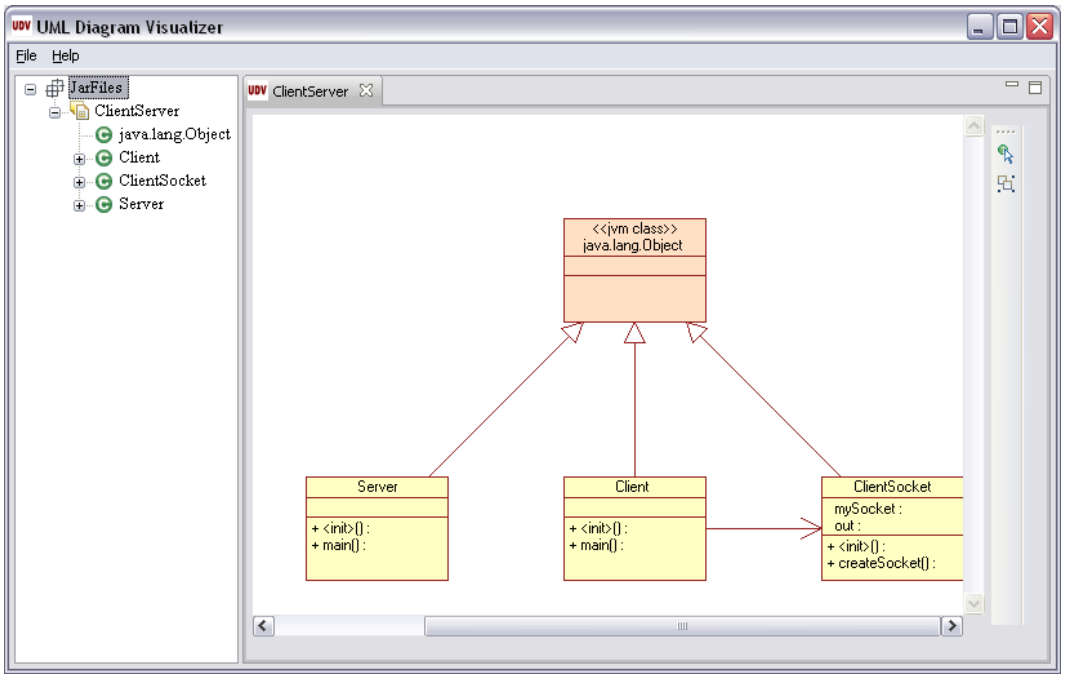


Figure 3.11. Screenshot 2: UML Class Diagram generated for Client-Server application.

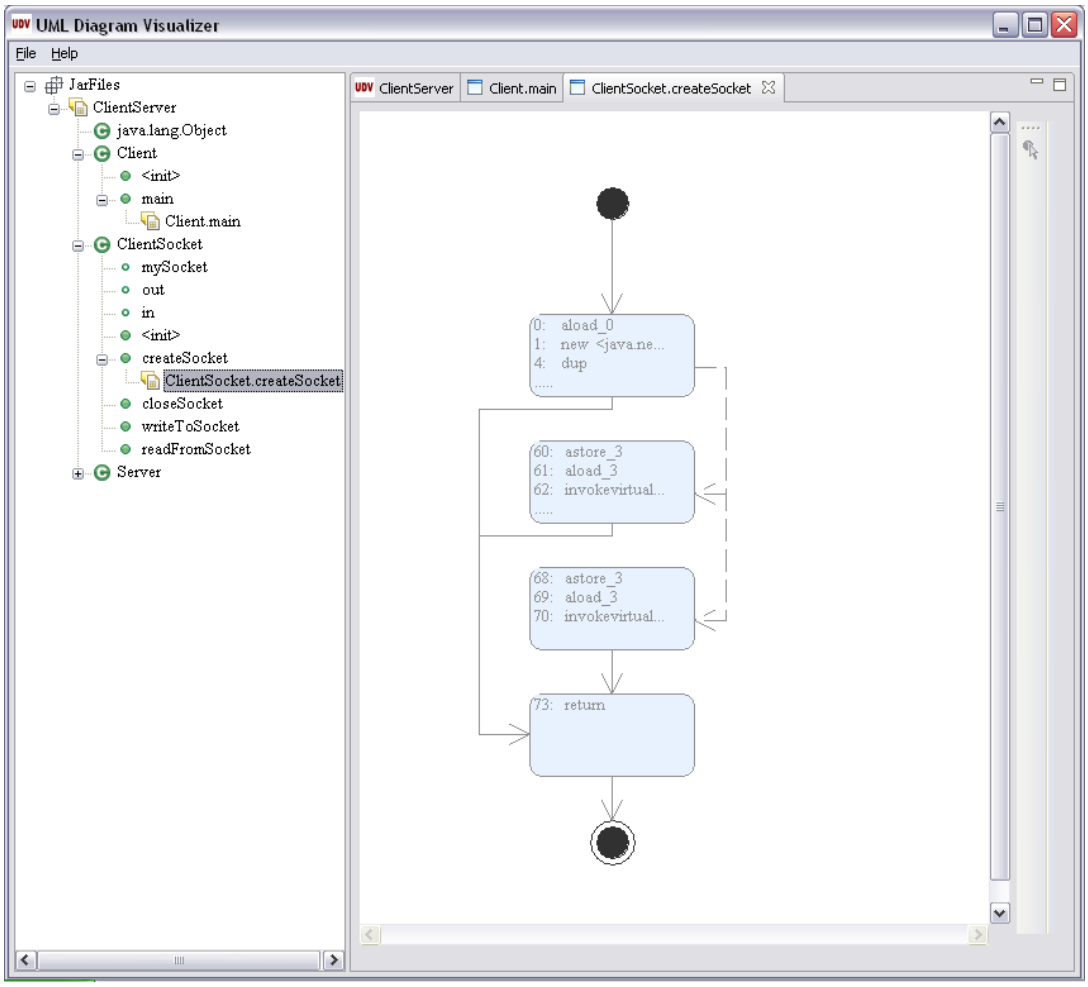


Figure 3.12. Screenshot 3: UML Activity Diagram generated, showing exception handlers.

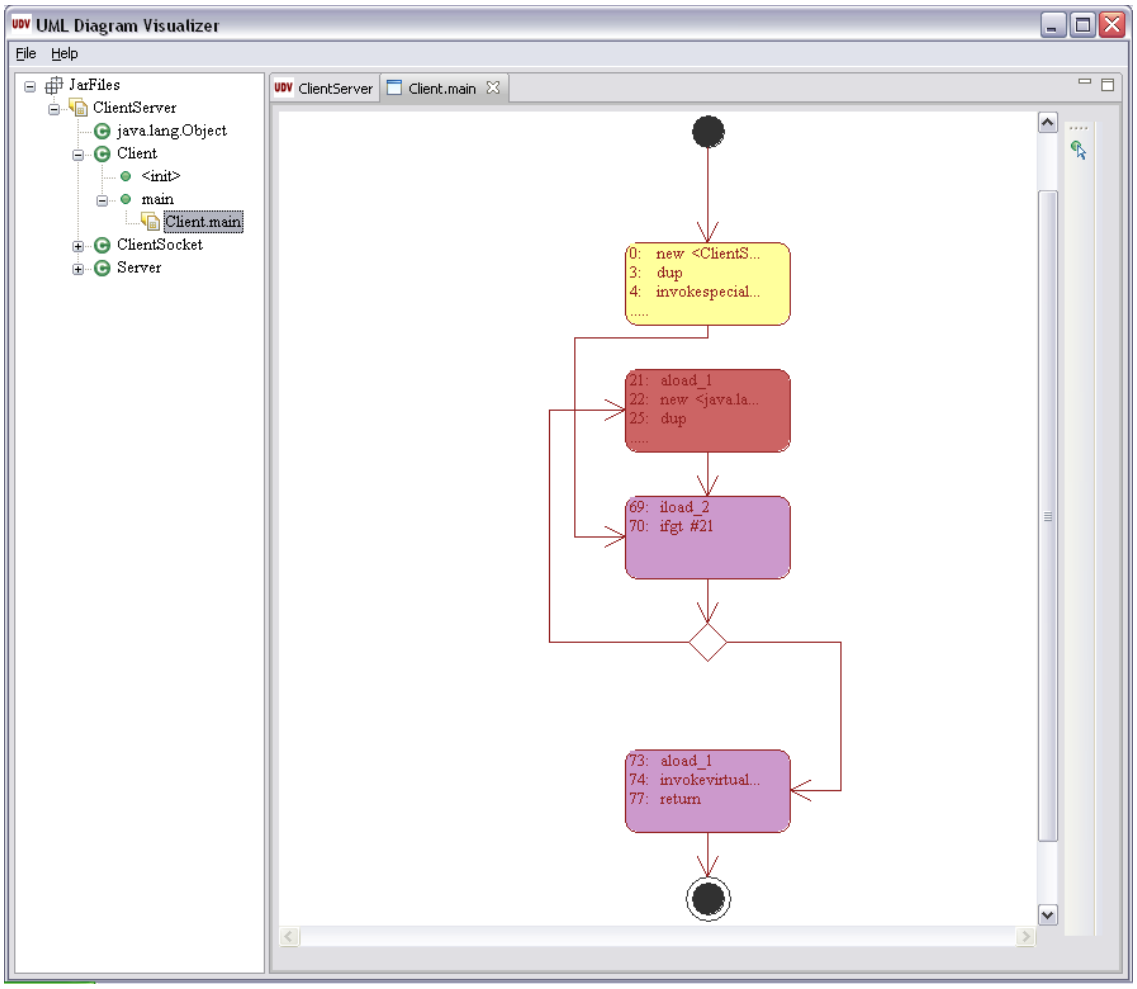


Figure 3.13. Screenshot 4: UML Activity Diagram generated, with all possible control flows mapped to a security policy.

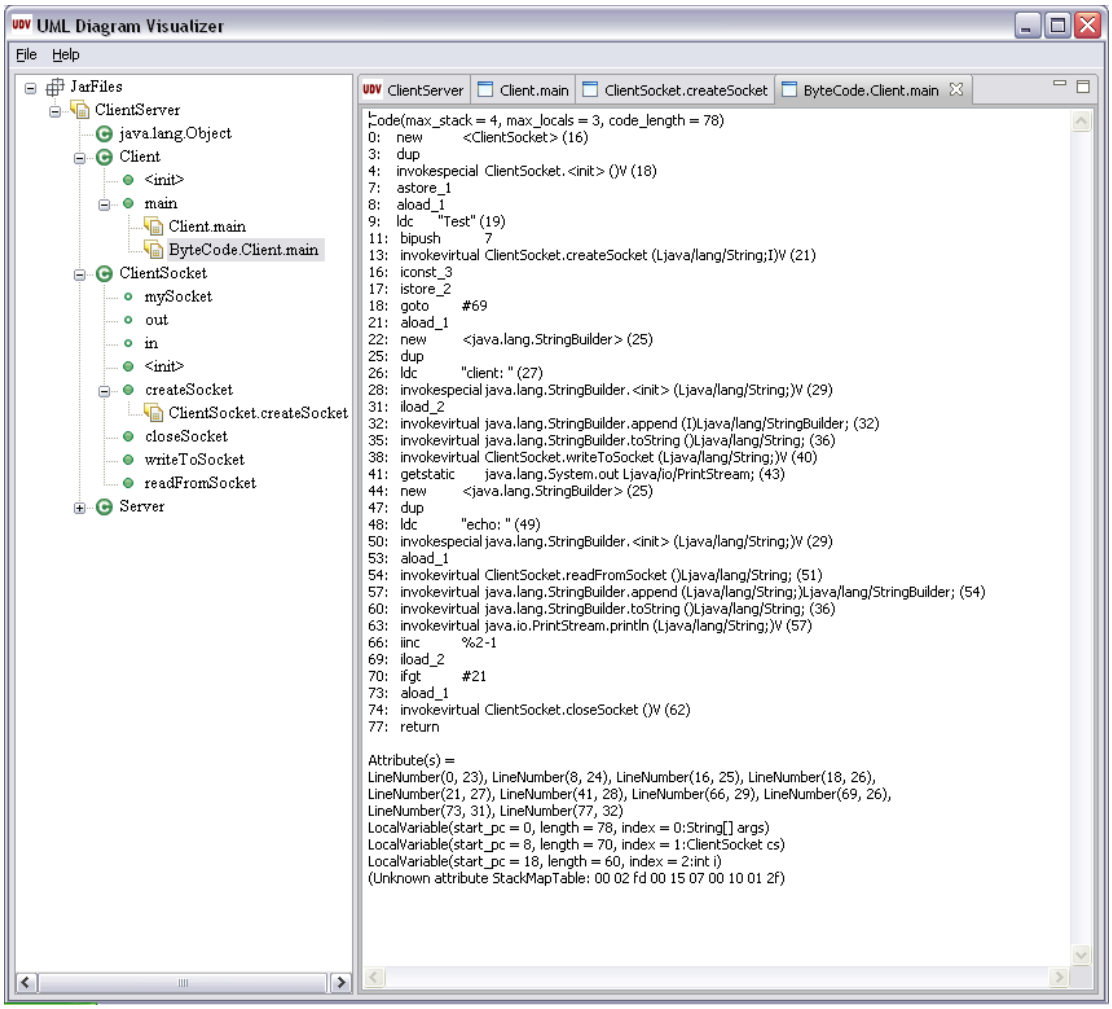


Figure 3.14. Screenshot 5: ByteCode view for viewing the underlying bytecode.

CHAPTER 4

VALIDATION

The visualization framework was validated using test applications. The static views engine as first tested using a medium-sized file-browsing application consisting of 17 classes. The visual class model produced by the static views engine was manually validated against the original source code of the test application.

We then validated the security-aware features of the static view using a test application that divulges a confidential file by sending its content over the network. We used the SPoX IRM system [6] to enforce a policy that prohibits write-access to the Java Socket library once a confidential file has been accessed. The class structures of the original, unsafe application bytecode and the SPoX-modified bytecode were then compared using the visualization framework. The new security class injected by the IRM was identified and highlighted in the visual model (see Screenshot 1).

To validate the output of the dynamic views engine, we enforced the same policy on a smaller, hand-written client-server application whose control-flow structure is simple enough to allow manual inspection. (see Appendix B for details of the specification of the security policy) The compiled bytecode was then used to generate UML diagram views using our framework. These diagram views were manually validated against the application class structures and control-flows. After the policy was applied, our visualizer generated security-aware control flows for the client application. These were manually inspected to confirm that

policy-violating instructions (i.e., network-send operations that could execute after file-read instructions) were correctly identified by the visual model (see Screenshot 3).

CHAPTER 5

CONCLUSION AND FUTURE WORK

We have introduced a security-aware, bytecode visualization framework that facilitates fast and easy prototyping and analysis of IRM security policies and their implementations. This is the first approach to address this concern. Experiments show that the framework, implemented as a prototype tool, can generate visual diagrams and identify code points where possible security violations could occur at runtime. Without an automated tool, such analyses are extremely difficult and time-consuming even for experts, since they involve manually understanding an application's binary structure, its possible control-flows, and its potentially security-relevant operations. Rapid development and prototyping of candidate security policies is therefore typically impractical without such a tool.

A number of limitations have been identified in the visualization framework. With respect to the UML diagrams created, for example, the framework does not support reverse engineering to the complete, standard definitions of UML class and activity diagrams. The current subset has been adequate for the example applications used in the validation, but may need to be extended in the future. In addition, the securityaware color coding scheme used has not been rigorously defined as a UML extension, such as a profile or a stereotype. The graph based algorithms to create the diagrams also need to be systematically defined and analyzed.

The approach taken to identify the possible security violations may include false positives, wherein some unreachable control-flows are identified as possible sources of policy

violations. However, the approach does not suffer from false negatives, conservatively detecting all possible security violations. The framework can be easily extended to provide further support for visualization with respect to security, including extended debugging functionality and support for visual modeling. These are avenues we intend to explore in future work.

APPENDIX A

Source code for Client.java from test Client-Server Application:

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class Client {

	public static void main(String[] args) {
		ClientSocket cs = new ClientSocket();
		cs.createSocket("Test", 7);
		int i = 3;
		while(i>0){
			System.out.println("echo: " + read("C:/Data/test.txt"));
			cs.send("client: "+i);
			i--;
		}
		cs.closeSocket();
	}

	public static String read(String path) {
		String contents = null;
		try {
			File file = new File(path);
			BufferedReader input =  new BufferedReader(new
FileReader(file));
			String line = null;
			while((line=input.readLine())!=null){
				if(contents==null){
					contents = line+"\n";
				}
				else{
					contents.concat(line+"\n");
				}
			}
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
	    return contents;
	}

}
```

Source code for ClientSocket.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.UnknownHostException;

public class ClientSocket {
	Socket mySocket = null;
	PrintWriter out = null;
	BufferedReader in = null;

	public void createSocket(String host, int port) {
		try {
			mySocket = new Socket(host, port);
			out = new PrintWriter(mySocket.getOutputStream(), true);
			in = new BufferedReader(new
InputStreamReader(mySocket.getInputStream()));
		}
		catch (UnknownHostException e) {
			e.printStackTrace();
		}
		catch (IOException e) {
			e.printStackTrace();
		}
	}

	public void closeSocket() {
		try {
			out.close();
			in.close();
			mySocket.close();
		}
	catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}

	public void send(String data) {
		out.println(data);
	}

	public String receive() {
		String data=null;
		try {
			data = in.readLine();
		}
	catch (IOException e) {
			e.printStackTrace();
		}
		return data;
	}
}
```

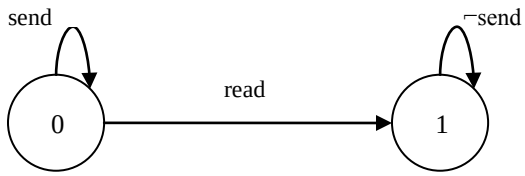
Bytecode for Client.main from test Client-Server Application:

```
Code(max_stack = 4, max_locals = 3, code_length = 79)
0:   new            <ClientSocket> (16)
3:   dup
4:   invokespecial  ClientSocket.<init> ()V (18)
7:   astore_1
8:   aload_1
9:   ldc            "Test" (19)
11:  bipush                 7
13:  invokevirtual  ClientSocket.createSocket (Ljava/lang/String;I)V (21)
16:  iconst_3
17:  istore_2
18:  goto           #70
21:  getstatic              java.lang.System.out Ljava/io/PrintStream; (25)
24:  new            <java.lang.StringBuilder> (31)
27:  dup
28:  ldc            "echo: " (33)
30:  invokespecial  java.lang.StringBuilder.<init> (Ljava/lang/String;)V (35)
33:  ldc            "C:/Data/test.txt" (38)
35:  invokestatic   Client.read (Ljava/lang/String;)Ljava/lang/String; (40)
38:  invokevirtual  java.lang.StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder; (44)
41:  invokevirtual  java.lang.StringBuilder.toString ()Ljava/lang/String; (48)
44:  invokevirtual  java.io.PrintStream.println (Ljava/lang/String;)V (52)
47:  aload_1
48:  new            <java.lang.StringBuilder> (31)
51:  dup
52:  ldc            "client: " (57)
54:  invokespecial  java.lang.StringBuilder.<init> (Ljava/lang/String;)V (35)
57:  iload_2
58:  invokevirtual  java.lang.StringBuilder.append (I)Ljava/lang/StringBuilder; (59)
61:  invokevirtual  java.lang.StringBuilder.toString ()Ljava/lang/String; (48)
64:  invokevirtual  ClientSocket.send (Ljava/lang/String;)V (62)
67:  iinc           %2        -1
70:  iload_2
71:  ifgt           #21
74:  aload_1
75:  invokevirtual  ClientSocket.closeSocket ()V (65)
78:  return
```

APPENDIX B

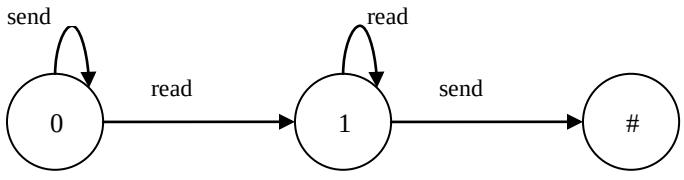
Security policy specification:

The security policy input to the visualizer is converted to a simplified non-deterministic version of the original policy. The original security automaton for the security policy that prohibits information leakage over the network is given as below.



Security automaton for the prototype policy used

The simplified deterministic version of the security automaton, introduces an error state # representing the occurrence of the specified security violation.



Deterministic automaton for the security policy

The text-based version of the automaton that is input to the tool: The first four lines of the text-based specification define the security states of the automaton and the corresponding color code to be used for each of them. The next four lines define the various security state transitions in the security automaton on the specific security-relevant operations of the bytecode.

```
0 255,255,156
1 204,255,156
-99 204,100,100
++ 204,153,204
0 0 call ClientSocket.send
0 1 call Client.read
1 1 call Client.read
1 -99 call ClientSocket.send
```

Text-based security policy for information leakage

REFERENCES

[1] T. Thibodeaux, .End user IT security training can save billions,. IndustryWeek, May 2009.

[2] J. Ligatti, L. Bauer, and D. Walker, .Run-time enforcement of nonsafety policies,. ACM Transactions on Information and System Security, vol. 12, no. 3, January 2009.

[3] F. B. Schneider, .Enforceable security policies,. ACM Transactions on Information and System Security, vol. 3, no. 1, pp. 30.50, February 2000.

[4] U. Erlingsson and F. B. Schneider, .SASI enforcement of security policies: A retrospective,. in Proc. New Security Paradigms Workshop, September 1999, pp. 87.95.

[5] F. Chen and G. Rosu, .Java-MOP: A monitoring oriented programming environment for Java,. Lecture Notes in Computer Science, vol. 3440, pp. 546.550, 2005.

[6] K. W. Hamlen and M. Jones, .Aspect-oriented in-lined reference monitors, . in Proc. ACM Workshop on Programming Languages and Analysis for Security, 2008.

[7] B. Gruegge and A. H. Dutoit, Object-Oriented Software Engineering: Using UML, Patterns and Java, 2nd ed. Prentice Hall, 2004.

[8] J. Arlow and I. Neustadt, UML 2 and the Uni_ed Process: Practical Object-oriented Analysis and Design, 2nd ed. Addison-Wesley, June 2005.

[9] P. Kouznetsov, .JAD: The fast Java decompiler,. http://www.kpdus.com/ jad.html.

[10] Ahpah Software, Inc., .The SourceAgain decompiler,. http://www.ahpah.com/products.html.

[11] The Eclipse platform,. http://www.eclipse.org.

[12] International Business Machines, .IBM software architect,. http://www-01.ibm.com/software/awdtools/swarchitect.

[13] H. M. Kienle and H. A. Müller, .Rigi: An environment for software reverse engineering, exploration, visualization, and redocumentation,. Science of Computer Programming, vol. 75, no. 4, pp. 247.263, April 2010.

[14] G. Conti, E. Dean, M. Sinda, and B. Sangster, .Visual reverse engineering of binary and data files,. Lecture Notes in Computer Science, vol. 5210, pp. 1.17, 2008.

[15] Byte code engineering library,. http://jakarta/apache/org/bcel/index.html.

[16] H. M. Kienle and H. A. Müller, .Requirements of software visualization tools: A literature survey,. in Proc. 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, June 2007.

[17] J. Seemann, .Extending the Sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams,. Lecture Notes in Computer Science, vol. 1353, pp. 415.424, 1997.

[18] Compilers- principles, techniques and tools, Alfred Aho, Ravi Sethi, Jeffery Ullman.

VITA

Aditi A. Patwardhan was born to Swati Patwardhan and Abhijit Patwardhan in Pune, India, where she did most of her schooling. In 2006, she graduated from the Univerysity of Pune with a Bachelors degree in Computer Science. Soon she joined Infosys Tecnologies ltd as an early career Software Engineer. After working at Infosys for about 22 months, she joined the University of Texas, Dallas in fall 2008 to pursue Master of Science in Computer Science. She is getting married to her fiancé Ashutosh Watway and plans to move to her hometown Pune with him.