Malicious Software Engineer Intrusion Detection Between Components

by

Snehadeep Sethia, B.E.

A THESIS

IN

COMPUTER SCIENCE

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

MASTERS OF SCIENCE

Approved

Dr. Michael (Eonsuk) Shin
Chairperson of the Committee

Dr. Yu Zhuang

Accepted

Dr. Ralph Ferguson
Dean of the Graduate School

December, 2010
Copyright 2010, Snehadeep Sethia
I dedicate this thesis
To my parents Surendra Sethia and Kanta Sethia; they are the reason why I am here
   To my friends for their love and support through thick and thin times
   To Dr. Shin who has supported me with his positive attitude and polite nature
ACKNOWLEDGEMENTS

There are a number of people without whom this thesis might not have been written, and to whom I am greatly indebted.

I owe my deepest gratitude to Dr. Michael Shin, the advisor of this thesis. I am inarticulate to show him my gratitude and appreciation for whatever he has done for me and this thesis. He has made available his support in a number of ways. He is a great advisor and a great mentor. I thank him for his patience that he has shown right from the beginning when I used to meet him with a lot of stupid ideas till the end of this thesis. His patience in showing me the proper direction when I seemed to be lost has proved to be very beneficial for me as well as for my thesis.

I thank my committee member and thesis reader, Dr. Yu Zhuang, for giving his precious time for reading my thesis and suggesting improvements in my thesis.

I am grateful to the Department of Computer Science at Texas Tech University for giving me an opportunity to pursue graduate studies.

I am grateful to the Department of IT Assessment for helping me with my finances by providing an on-campus job.

It is a pleasure to thank all my friends for their support and encouragement.
# Table of Contents

ACKNOWLEDGEMENTS .................................................................................................................. iii

ABSTRACT ...................................................................................................................................... vi

LIST OF TABLES ............................................................................................................................... vii

LIST OF FIGURES ............................................................................................................................. viii

1. INTRODUCTION ............................................................................................................................. 1

2. RELATED WORK .............................................................................................................................. 4
   2.1 Introduction ................................................................................................................................ 4
   2.3 Intrusion Detection using Sequence of System Calls [7] .................. 6
   2.5 Design of Secure Components for Internal Intrusion Detection [6] ... 9
   2.6 Using Model Checking to Identify Errors in Intrusion Detection Signatures [8] ............................. 10
   2.7 Model Checking One Million Lines of C Code [9] .......................... 11
   2.8 Intrusion Detection via Static Analysis [10] .................................... 12
   2.10 A Temporal Logic Based Framework for Intrusion Detection [12] ... 13
   2.11 Conclusions .............................................................................................................................. 13

3. PROBLEM STATEMENT AND RESEARCH APPROACH ......................................................... 14
   3.1 Introduction ............................................................................................................................... 14
   3.2 Motivation ............................................................................................................................... 14
   3.3 Problem Statement ................................................................................................................... 15
   3.4 Research Approach .................................................................................................................. 15
   3.5 Summary ................................................................................................................................ 16

4. MALICIOUS SOFTWARE ENGINEERING INTRUSION DETECTION ............................. 17
   4.1 Introduction ............................................................................................................................... 17
   4.2 Design of System Detector for Malicious Software Engineer Intrusion Detection between Components .................................................................................................................. 17
   4.3 Authentication of Message Communication between Component and System Detector .......................................................................................................................... 20
   4.4 Centralized System Detector .................................................................................................... 22
   4.5 Distributed System Detector .................................................................................................... 26
   4.6 Implementation and Evaluation the Proposed Approach ......................... 32
   4.7 Summary ................................................................................................................................ 39

5. CASE STUDY – AUTOMATED TELLER MACHINE SYSTEM ................................................. 40
5.1 INTRODUCTION ................................................................................................................. 40
5.2 PIN VALIDATION USE CASE ............................................................................................ 41
5.3 WITHDRAW FUNDS USE CASE ......................................................................................... 46
5.4 TRANSFER FUNDS USE CASE ......................................................................................... 51
5.5 QUERY ACCOUNT USE CASE ......................................................................................... 56
5.6 CONSOLIDATED SYSTEM STATE MACHINE ................................................................. 61
5.7 SUMMARY .......................................................................................................................... 62

6. CASE STUDY – ELECTRONIC COMMERCE SYSTEM .................................................. 63
   6.1 INTRODUCTION ............................................................................................................... 63
   6.2 PLACE REQUISITION USE CASE .................................................................................. 64
   6.3 PROCESS DELIVERY ORDER USE CASE .................................................................... 70
   6.4 CONFIRM SHIPMENT USE CASE .................................................................................. 72
   6.5 CONFIRM DELIVERY USE CASE .................................................................................. 76
   6.6 SEND INVOICE USE CASE ......................................................................................... 79
   6.7 CONSOLIDATES SYSTEM STATE MACHINE ............................................................. 86
   6.8 SUMMARY .......................................................................................................................... 88

7. CONCLUSIONS AND FUTURE WORK ............................................................................. 89

REFERENCES .......................................................................................................................... 90
ABSTRACT

This thesis describes an approach to detecting malicious software engineer intrusion between components in application systems using business processes (use cases) of applications. The approach detects malicious codes inserted by malicious software engineers to the system during the software development or the maintenance phase. This research extends a previous research about malicious software engineer intrusion within a component. The proposed approach detects intrusion using system detectors that are designed to encapsulate the relationships between components. Those relationships are represented with the UML state machines. The system detectors communicate with objects in components in order to monitor the communication between components in which the system detectors authenticate the messages from objects. This is to avoid fake messages from malicious code. The proposed approach has been applied to two case studies – Automated Teller Machine System and Electronic Commerce System – and the performance of the system detectors has been evaluated with case studies.
LIST OF TABLES

4.1 Performance Analysis of Withdraw Funds use case ........................................ 34
4.2 Performance Analysis of Transfer Funds use case ........................................ 35
4.3 Performance Analysis of Query Account use case ........................................ 36
4.4 Performance Analysis of Place Requisition use case .................................... 37
4.5 Performance Analysis of Process Delivery Order use case ............................. 38
LIST OF FIGURES

4.1 Message Communication between Objects in the ATM System and the ATM System Detector for Transfer Funds use case ..................................................19
4.2 State Machine in the ATM System Detector ........................................................................20
4.3 Distribution of Keys ...........................................................................................................22
4.4 Collaboration diagram for Withdraw Funds use case – Client Side ................................. 24
4.5 Collaboration diagram for Withdraw Funds use case – Server Side ................................. 25
4.6 Statechart for Withdraw Funds use case – System Detector .......................................... 26
4.7 Collaboration diagram for Send Invoice use case ............................................................. 29
4.8 State machine for Supplier Invoice Detector ................................................................. 30
4.9 State machine for Delivery Invoice Detector .................................................................... 31
4.10 State machine for Requisition Invoice Detector .......................................................... 32
4.11 Architecture diagram of a Secure ATM System ............................................................... 40
4.12 Collaboration diagram for PIN Validation use case – Client Side .................................. 41
4.13 Client state machine for PIN Validation use case ........................................................... 42
4.14 System Detector’s state machine for PIN Validation use case ......................................... 43
4.15 Collaboration diagram for PIN Validation use case – Server Side ................................. 44
4.16 Bank Server Monitor’s state machine for PIN Validation use case .................................. 45
4.17 Collaboration diagram for Withdraw Funds use case – Client Side ................................ 46
4.18 Client state machine for Withdraw Funds use case ........................................................ 47
4.19 System state machine for Withdraw Funds use case ...................................................... 48
4.20 Collaboration diagram for Withdraw Funds use case – Server Side ................................ 49
4.21 Server state machine for Withdraw Funds use case ....................................................... 50
4.22 Collaboration diagram for Transfer Funds use case – Client Side ................................... 51
4.23 Client state machine for Transfer Funds use case .......................................................... 52
4.24 System state machine for Transfer funds use case ........................................................ 53
4.25 Collaboration diagram for Transfer Funds use case – Server Side ................................. 54
4.26 Server state machine for Transfer Funds use case .......................................................... 55
4.27 Collaboration diagram for Query Account use case – Client Side .................................. 56
4.28 Client state machine for Query Account use case .......................................................... 57
4.29 System state machine for Query Account use case ........................................................ 58
4.30 Collaboration diagram for Query Account use case – Server Side ................................ 59
4.31 Server state machine for Query Account use case .......................................................... 60
4.32 Consolidated System state machine ............................................................................... 61
4.33 Architecture diagram of a E-Commerce System .............................................................. 63
4.34 Collaboration diagram of Place Requisition use case ...................................................... 64
4.35 State machine for Customer Requisition Detector .......................................................... 65
4.36 State machine for Customer Delivery Detector .............................................................. 66
4.37 Collaboration diagram of the Requisition System .............................................................. 67
4.38 State machine for the Requisition Monitor in the Requisition System ............................ 68
4.39 Collaboration diagram of the delivery System ................................................................ 69
4.40 State machine for the Delivery Monitor in the Delivery System .................................... 69
4.41 Collaboration diagram for Process Delivery order use case .......................................... 70
4.42 State machine for Supplier Delivery Detector ............................................................... 72
CHAPTER 1
INTRODUCTION
A number of intrusion detection systems have been developed for improving computer system security, but application systems are still vulnerable to the attacks from insider intrusions as well as outsider intrusions of the systems. Most of the intrusion detection systems have been focused on outsider intrusion detection for operating systems, network systems and database systems, and less attention has been paid to intrusion detection for application systems, such as banking systems and finance systems, which contain lots of business security information. Most of the intrusion detection approaches for application systems focuses on outsider intrusion, but not much is available for insider intrusion, such as malicious codes created by malicious software engineers [1].

According to the Insider Threat Study team composed of U.S. Secret Service (USSS) behavioral psychologists and CERT information security experts, technical employees such as software engineers have taken advantage of loopholes left in the software development life cycle (including the system maintenance phase) to deliberately perform malicious technical actions. The Insider Threat Study team collected approximately 150 insider threat cases that occurred in US critical infrastructure sectors between 1996 and 2002, and examined them from both technical and behavioral perspectives. Based on findings from the analysis of insider threat cases, the Insider Threat Study team suggests that (1) organizations that develop and/or maintain application systems must consider insider threats in the software development life cycle, and (2) changes to systems and applications must be controlled to prevent insertion of malicious code or programs [1].

While many organizations use mandatory code review for development of systems and follow stringent configuration processes for change control, several insiders such as software engineers are able to inject malicious code into new systems or existing systems. This is because, although review of all the program codes in a system can detect malicious code or many different defects (used by insiders to insert malicious code to a
system) using the domain or programming language knowledge, line-by-line review of the entire code is not cost and time effective for projects that have a short timeline to be met. Also ineffective configuration or change control processes contributed to insiders’ ability to insert malicious code to the systems as well. Practically, code review or change control process can be limited to only part of programs that require security in the system [1].

As security is becoming more important for business these days, secure application systems are required to support business security. In general, application systems are developed and maintained by multiple software engineers, such as program developers and maintenance software engineers. Though most of software engineers are ethical, a few may be unethical and malicious. A malicious software engineer, who may not be satisfied with the organization, may add malicious code to application systems or change existing code maliciously such that the malicious code is activated either periodically or non-periodically for personal gain or disrupt service or destroy the organization’s system and/or data. This is referred to as malicious software engineer intrusion in this thesis.

A previous research provides an approach to detecting malicious software engineer intrusion where malicious code within a component gains or changes confidential information encapsulated in security-relevant objects of the component. To protect security-relevant objects within a component from the malicious code, the research describes about how to design secure components in terms of state-dependent components and state non-dependent components. However, the previous research does not tackle another type of malicious software engineer intrusion where malicious code within a component or outside a component accesses maliciously security-relevant objects in other different components. Therefore it is necessary for an approach to detecting this type of malicious software engineer intrusion.

This thesis describes an approach to detecting the malicious software engineer intrusion during runtime where malicious code within a component or outside a component
accesses security-relevant objects in different components. To detect this type of malicious intrusion, the accesses to security-relevant objects in a component from other objects in different components are monitored by security detector objects to verify security requirements in terms of malicious software engineer intrusion. A security detector object is used in a client-server application system and a distributed application system.

This thesis is organized as follows: Chapter 2 describes existing approaches to analyze intrusion detection. Chapter 3 states the problem statement and the research approach. Chapter 4 describes the details of detection of compromised security between software components. Chapter 5 and 6 illustrate ATM system and Electronic commerce system (B2B) case studies respectively. Finally, the thesis is concluded in Chapter 7.
2.1 Introduction

Modern computer systems are prone to many intrusions [2]. Any action or a set of actions which can either be external or internal, that attempt to compromise the integrity, confidentiality, accountability or availability of a resource is intrusion [3]. An intruder is a person/entity who has/have or is trying to gain unauthorized access to the system. There are two types of intruders, external intruders and internal intruders. External intruders are those who do not have any authorized access to the system. Internal intruders are those who possess some authorized access to the system they attack.

Internal intruders are further classified as Masqueraders, Legitimate intruders and Clandestine intruders. External intruders who have penetrated the system successfully and are pretending as an authorized entity are called masqueraders. Intruders who misuse the access of system and/or data are legitimate intruders. Intruders who have obtained administrative control and either operate below the level of auditing or block themselves from being audited by misusing their privileges are clandestine intruders [4].

The primary concern of intrusion detection is to detect illegal activities and acquisitions of privileges that cannot be detected by the flow of information and authorization models [5]. Intrusion detection systems (IDS) maintain the Confidentiality, Integrity, Availability, and Accountability of a system. Intrusion detection has mostly been performed at the Operating System (OS) level by comparing observed system resource usage against the expected system resource usage. OS intrusion detection systems seldom catch internal intruders because they are legitimate users of the system, and they do not deviate significantly from expected behavior. Internal intrusion can be detected by using application specific intrusion detection systems that uses the semantics of the application [4].
Currently intrusion detection is performed using two approaches, Anomaly Detection and Misuse Detection (Signature-based detection) [5]. Anomaly detection based IDS build an acceptable dataset and activities are compared with this dataset. Any activity that deviates from this dataset is considered abnormal and is flagged for exceptions [6]. Tripwire, NIDES IDS fall under this category [4]. Misuse detection based IDS build a dataset of patterns. Patterns are known ways of penetrating a system. Any activity that matches a pattern from this dataset is considered abnormal and is flagged for exceptions [4]. MIDAS and STAT IDS fall under this category [4].

2.2 Application Intrusion Detection using Language Library Calls [3]

Library call signatures are defined to be sequences of library call invocations. The first step is to build a signature database. A signature database can be built in two ways, namely synthetic signature database and real signature database. The synthetic signature database is built by running the application several times in its normal mode of usage. While running the application, a live user environment is used to build the real signature database. In both the cases signatures are build by sliding a window of size k across the library calls made by the application and recording each unique sequence of length k that is encountered.

Each application has a unique signature database. Each signature database is specific to application code, hardware architecture, software version and configuration, local administrative policies, and usage patterns of the users. A robust signature database once constructed for a particular application can be used to monitor the ongoing behavior of the processes that was created by that application.

Monitoring for deviant behavior is done using the same technique that was used to build the signature database. The application behavior is considered to be anomalistic if the monitored sequences of length k vary profoundly from those in the signature database.
There are different approaches to detect anomalies. One way is to record the mismatch count. Monitored sequences that are absent in the signature database are mismatches. The mismatch count is the summation of mismatches. Another method is to record the percentage of mismatches. It takes at most $k$ comparisons to determine a match or a mismatch for a new sequence of length $k$. Another approach is the locality frame count. Locality frame count assumes that anomalous sequences will occur in local bursts. When a process is under attack, there may be a short period of time (known as locality) when the percentage of anomalous sequences is much higher. For example, fifteen anomalies within a short period of time are of greater concern than over a long period of time. The number of sequences that are considered to be local to one another is called the size of the locality frame.

The disadvantages of this approach are; a real signature database may cover all the possible paths as a synthetically built signature database. When building a real signature database it’s difficult to ensure that no intrusion occurred during database generation. Also, in a real execution environment, there is no mechanism to force execution of all possible paths through the application code. How many mismatches does it take to indicate truly anomalous behavior? How does locality frame count work if intrusion doesn’t occur in limited interval of time? How do we detect intrusion between applications/modules?

2.3 Intrusion Detection using Sequence of System Calls [7]

In this approach the program under observation for anomalistic behavior is considered as a black box, which emits some results when run. The method has two phases for intrusion detection. In the first phase profiles or databases of normal behavior are built (this is similar to training a system); in the second phase these databases are used to monitor system behavior for significant deviations from normal behavior (similar to testing a system).
A very simple method is used to build the normal databases. A normal database consists of traces of unique sequence of length k system calls that are generated by a particular process. Each program has a unique database, which depends on and is unique to a particular architecture, software version and configuration, local administrative policies, and usage patterns of the users. For a given program, once a stable database is constructed, it can be used to monitor the ongoing behavior of the processes invoked by that program.

Monitoring the deviant behavior uses the same technique that was used to generate the database. All overlapping sequences of length k in the new trace are observed against the ones stored in the normal database. Sequences that do not match with any stored in the normal database are considered to be mismatches. There are possibilities that the normal database is incomplete and does not cover all the possible paths of normal behavior. One solution is to count the number of mismatches occurring in a trace, and only consider it as anomalous if the mismatch count crosses a predefined threshold value.

The system works on mismatch count crossing a threshold to determine an intrusion. What is the optimal value for this mismatch count? What if the intruding process has a sequence of system calls within the tree? What if the intrusion sequence is not present in the normal database and occurs less than the threshold value? How do we detect intrusion between applications/modules?

2.4 A Pattern Matching Model for Misuse Intrusion Detection [5]

One of the approaches to intrusion detection is misuse detection. The approach used is a variation of misuse detection, state transition analysis, by using pattern matching to detect system attacks. When patterns are matched any user-specified actions associated with the pattern gets executed. Along with associated actions, patterns also have pre and post conditions associated with them that must be satisfied before and after the match.
Invariants may also be a part of a pattern that specifies a condition, which needs to be satisfied while matching a pattern.

The pattern matching used is based on the outlook of an event. Events are legitimate changes in the state/part of the system. An event can be a single action or series of actions resulting in a single observable record by a user or system. A primary requirement of applying pattern matching to intrusion detection is that matching be done with follows semantics rather than immediately follows semantics. For example, with follows semantics the pattern xy specifies the occurrence of the event x followed by the occurrence of event y. It does not represent x immediately followed by y with no intervening event. This means that any two adjacent sub patterns within a pattern are implicitly separated by an arbitrary number (positive zero) of events of any type. This assumption is appropriate in many systems.

Advantages of this approach are that it is portable, intrusion signatures can be reused across different vendor’s implementation. Signatures can also be easily ported to systems with different policies and ratings. Dynamic addition of the signatures to the matching engine is possible while maintaining the partial matches of signatures that are already present. The disadvantage of this technique is that the newly added patterns cannot be optimized with respect to the already compiled patterns in the engine. Signatures can be prioritized by considering each token as a thread of control. Each thread then fetches events from an event manager and acts on them. By prioritizing certain threads, patterns can be prioritized for matching.

Other difficulty in intrusion detection using pattern matching is the rate at which the data generated by modern processors must be matched. Another major problem is the nature of the matching itself. An attacker may perform several actions disguising his/her identity, ultimately leading to a system compromise. The complexity of matching increases rapidly with increasing complexity of signatures. This approach doesn’t address intrusion detection between components of an application system.
2.5 Design of Secure Components for Internal Intrusion Detection [6]

Intrusion detection within a component is done in either a state-dependent secure component or a non-state dependent secure component. The intrusion is detected using a finite state machine described in a statechart in UML. This finite state machine consists of both security states and application specific states. The state machine within the secure component transitions from one state to another on receiving messages from different entities within that component. Alarms are generated when the transition of the state machine does not follow the normal path of execution.

A state-dependent control object is a control object whose behavior varies in each of its state and a state-dependent component has one or more of these. A state-dependent control object receives incoming messages/events that cause state transitions and generates output events that control other objects. The current state and the incoming message/event decide the output of a state-dependent control object. The original state machine in a state-dependent control object of a state-dependent component contains only application states of a system, whereas the extended state machine for a secure state-dependent component has security states along with the application states. The extended state machine transits from a secure state to another on legitimate invocation of operation on a security-relevant object by other objects within the secure state-dependent component.

A secure state-dependent component detects intrusion using the extended state machine of a state-dependent control object in the component. At a given state, a state-dependent control object expects to receive one of the expected messages to transit from one secure state to another in the extended state machine. Whenever an operation of a security-relevant object is invoked by other object, it notifies the invocation to the state-dependent control object. These notification messages transits the extended state machine for the state-dependent control object if they are one of the expected messages amongst the set of
messages. However, when a state-dependent control object receives an unexpected message from a security-relevant object, it detects that intrusion has occurred in the security-required object.

A non-state dependent component is a stateless component, which does not need to maintain the component state so as to provide services. A non-state dependent component does not contain any state-dependent control object within it. Thus a security monitor encapsulating the state machine is used to monitor the non-state dependent component. A security monitor object receives incoming messages/events that cause state transitions. The state machine in a security monitor object of a non-state dependent component has security states. The state machine transits from a security state to another on legitimate invocation of operation on a security-relevant object by other objects within the secure non-state dependent component. Similar to a secure state-dependent component, a secure non-state dependent component detects intrusion using the state machine of a security monitor object which lives within that component.

The disadvantage of this approach is that it addresses intrusion detection within a component and not in between components.

2.6 Using Model Checking to Identify Errors in Intrusion Detection Signatures [8]

The approach used makes use of EDL (Event Description Language), which follows colored Petri nets. The signatures modeled using this technique are more detailed than any other. A typical signature in EDL would have Places and Transitions which are connected by directed edges. System states where the related attack has to traverse are called Places, while Transitions are caused by events which lead to change of state. An attack which is execution of a signature is represented by a token which flows from state to state.

Initial, Interior, Escape and Exit are the four type of places described in the EDL. The starting places of a signature are Initial places; the ending place of a signature is the Exit
place. If a token reaches the end place, it means that the attack is/was executed successfully. Escape places stop the analysis of an attack making the attack impossible to complete. The rest are all Interior places.

Though EDL is a very intuitive signature language, newly modeled signatures can contain errors like unreachable places or transition conditions. Other possible errors could be missing token flows between places. These limitations are overcome by transforming the EDL semantics into Promela specifications. The limitations of Promela specifications are that it addresses attacks or intrusions within a component and not in between components.

2.7 Model Checking One Million Lines of C Code [9]

This technique uses MOPS, which is a compile time analysis tool. For a given program and its security property, MOPS checks whether the program violates the security property. Programs have temporal safety properties, i.e., security-related operations are to be executed in certain order. MOPS checks temporal safety properties of a program using Finite State Automation.

Pushdown model checking technique is used by MOPS to check if a program violates a temporal safety property. The control flow graph of a program is searched exhaustively to check if any path violates a safety property. Pushdown model checking addresses searching of inter-procedural in a context-sensitive manner. If MOPS finds violations it generates the program paths that caused the violation.

The disadvantages are that MOPS is not suitable to check violations in concurrent programs. It can address violations in a specific program/module but cannot address violations that can occur in between components/modules.
2.8 Intrusion Detection via Static Analysis [10]

The approach is to build a model of expected behavior of an application, which is built statically from program source code. The model is built using the application source code, along with a fixed model for the underlying operating system. During modeling, a set of system calls that the application can ever make is created. If at runtime the application makes system calls that are not in this model, execution of the application is stopped and alarms are generated.

The disadvantages of this system are that while modeling it might not be possible to trace all possible paths. The system can be fooled by mimicry attack, where the intruder behaves most exactly as the application should. The system addresses attacks within a component/application but doesn’t take care of attacks in between components in a distributed application.

2.9 Intrusion Detection through Dynamic Software Measurement [11]

Every program performs certain functionality and operations. Every operation that is executed creates a subset of software modules to execute which in turn creates a distinct signature of transition events. The approach used tracks real time intrusion detection. The approach not only tracks the external events produced by the system but also the internal behavior of the software. A set of known profiles at each module level is defined before. Each application and user creates a unique signature that is matched against these set of profiles at run time. If the behavior under observation deviates from the accepted one then alarms are generated.

The disadvantages here are the profiles need not cover all possible paths of execution of a module thus generating a lot of false positives. The system addresses intrusion in a module level but doesn’t address at a component or in between modules of an application.
2.10 A Temporal Logic Based Framework for Intrusion Detection [12]

The proposed intrusion detection framework is based on run time monitoring of temporal logic specifications. Intrusion patterns are specified in EAGLE. EAGLE stands for Expressive Monitoring Temporal Logic. The logic is to monitor the activity using an online algorithm; this algorithm matches the activity against the system execution traces and generates alarms whenever specifications are violated. An example attack scenario would be password guessing attack. An attack would be detected when the attempt to guess the password crosses a threshold value from a particular IP address. The system also keeps track of the number of times the threshold has crossed in a small duration of time from the same IP to flag it as an attack.

The disadvantage is that the system would generate false alarms if a genuine user has forgotten his/her password and is trying to guess it. The approach doesn’t address intrusion detection in between component of an application.

2.11 Conclusions

This chapter shows the different approaches, their advantages and limitations to intrusion detection.
CHAPTER 3

PROBLEM STATEMENT AND RESEARCH APPROACH

3.1 Introduction
This chapter describes the problem statement for this research and the research approach. Section 3.2 describes the motivation. Section 3.3 describes the problem statement. The research approach is described in Section 3.4. Section 3.5 summarizes this chapter.

3.2 Motivation
A malicious ATM software engineer (developer or maintenance software engineer) may add an extra object to either inside or outside a component so that she/he accesses security information within different components maliciously. Figure 2 depicts the UML communication diagram for the Transfer Funds service in the ATM Client component, which is simplified for this thesis. The message sequence K1 through K6 receives a customer input and requests transfer funds from the ATM Server component. A Malicious Object is designed to invoke the read() operation of ATM Transaction object that has the customer transaction information. (This is a case of violation of the confidentiality security requirement). The Malicious Object may request transfer funds maliciously from the ATM Server component (This is also a case of violation of the integrity security requirement).

A malicious ATM software engineer also may create a malicious object outside the ATM Server and ATM Client components, which invokes some operation of objects within the ATM Server or Client components maliciously. The malicious object may invoke the read() operation of ATM Transaction object in the ATM Client component or may invoke the credit() operation of Savings Account object in the ATM Server component.
3.3 Problem Statement

A previous research provides a detection approach against malicious software engineer intrusion, where malicious code added to a component gains or changes security information encapsulated in security-relevant objects in the component. However, the approach does not address malicious software engineer intrusion in which malicious code in a component maliciously accesses security-relevant objects in different components. Similarly, malicious software engineer can define a malicious object outside components and maliciously accesses security-relevant objects defined in components. These types of malicious software engineer intrusions need to be detected so that the security-relevant objects are protected from malicious access.

3.4 Research Approach

To resolve this problem, the following research approach has been taken:
i. Develop an approach to detecting malicious software engineer intrusions among components, and between malicious object and a security-relevant component.

ii. Validate this approach through case studies.

3.5 Summary

This chapter has described the problem statement and the research approach to solve the same.
CHAPTER 4
MALICIOUS SOFTWARE ENGINEERING INTRUSION DETECTION

4.1 Introduction
This chapter describes an approach to detect malicious software engineer intrusion between software components of an application system. A system detector is designed for this type of intrusion detection, encapsulating the state information between components. This approach generates alarms whenever there is a compromise in between software components of an application system. The system detector can be designed for both centralized application systems and distributed application systems.

This chapter is organized as follows. Section 4.2 addresses design of System Detector for malicious software engineer intrusion detection between components. Section 4.3 describes authentication of message communication between component and System Detector. Section 4.4 depicts the Centralized System Detector while the Distributed System Detector is shown in Section 4.5. Section 4.6 shows implementation and evaluation of the proposed approach and finally Section 4.7 summarizes this chapter.

4.2 Design of System Detector for Malicious Software Engineer Intrusion Detection between components
The intrusion detection methods for security-relevant (state-dependent or non-state dependent) components described in Nipul’s thesis [6] can detect only malicious software engineer intrusion when a malicious code in a component invokes some operation of a security-relevant object within the same component. However, some malicious code in a component can be designed to avoid this intrusion detection in such a way that it requests a service maliciously from a different security-relevant component. An example is that the Malicious Object in the ATM Client component in Figure 1 of Chapter 3 requests a Transfer Funds service from the security-relevant non-state dependent ATM Server.
component. In this case, the malicious transfer funds service request is processed by the security-relevant ATM Server component without detecting the intrusion.

A system detector is designed to detect malicious software engineer intrusion between different components. A system detector monitors the interaction between different components, detecting a malicious intrusion when a component requests a service maliciously from a different, security-relevant component. For this, the system detector contains the interaction relationship between components, which can be described using a state machine. When a service (that is, a use case) provided by a component is triggered by an internal or external event, the component notifies the system detector that traces the use case sequence. This use case sequence between components is described in a sequential or concurrent state machine. The system detector detects a malicious intrusion between components if a service request from a component does not take place as specified in the state machine.

The objects in components send notification messages to the system detector depending on the message communication styles such as asynchronous message communication or synchronous message communication. In case of asynchronous message communication, a service request object in a component sends a notification message to the system detector before requesting a service from another component, while a service provider object notifies the system detector on receiving a service request from other component. For synchronous message communication, objects send notification messages to the system detector before sending the result to a component and on receiving result from a component. To detect malicious code quickly, interface objects may need to notify the system detector when they receive input from other components or actors.

Figure 4.1 depicts an ATM system detector between the ATM Client and Server components, which encapsulates a state machine describing interaction between the components. When the Transfer Funds service (use case) is triggered by the ATM Customer, the Customer Interface notifies the ATM system detector of the Transfer Input
(K1.1), which causes a transition from the Waiting from Client state to the Preparing Transfer in Client state in Figure 4.2. The ATM Client component notifies the ATM system detector when it sends a message (K4.1 in Figure 4.1) to the ATM Server component as well. The Bank Transaction Server object in the ATM Server component also notifies the ATM system detector whenever it receives the Request Transfer message (K4.2 in Figure 4.1). The notification message causes a transition from the Preparing Transfer in Client state to the Transferring in Server state in the state machine of the detector. The Customer Interface and Bank Transaction Server are assumed secure. The Transfer Funds should be requested by the ATM Client component, and the request should take place when the ATM System detector is in the Preparing Transfer in Client state. Otherwise, The ATM System detector detects malicious software engineer intrusion. This is the case where the Malicious Object requests transfer funds maliciously from the ATM Server component. Similarly, when a reply is sent back to the ATM Client component from the ATM Server component, the components notify the ATM system detector.

**Figure 4.1** Message Communication between Objects in the ATM System and the ATM System Detector for Transfer Funds use case
4.3 Authentication of Message communication between component and System Detector

The messages between an interface object and a system detector are authenticated when the messages arrives at the system detector. The detector needs to check who sent the message in order to prevent a malicious object from sending a fake message to the system detector. A malicious object may go through the authentication of system detector if it has information about the details of authentication, such as the key used for authenticated data.

Asymmetric key cryptography is used for sender authentication. Asymmetric key algorithms are used to create private and public key pair which is mathematically related. The algorithm used for authentication is pre-decided by the System Detector and the individual components. The System Detector has a list of public keys of all the
components which interact with it. The individual objects within a component that interact with the System Detector have their private keys. An object within a component before sending messages to the System Detector encrypts it with its private key, on receiving this message the System Detector decrypts it using the public key of that component thus authenticating that the message is originated from a valid component.

To protect releasing key information for authentication, it is important to generate and distribute the keys securely. Executor is an object that is prevented from the malicious software engineer. Keys are generated by the executor object using the RSA public key algorithm. Along with the keys the executor generates UID’s (Unique Id’s). Once the individual components are installed the executor object distributes the keys and UID’s securely to the objects within components automatically. After the keys and UIDs are distributed automatically, the components should not be changed. In case where the components are changed, the Executor should generate keys and UIDs and distribute them to objects again. This is because the keys and UIDs are compromised by malicious software engineers. Figure 4.3 depicts the key distribution of an ATM System; the Executor object generates public-private key pairs and UIDs and distributes it securely to the Customer Interface, ATM Control, Bank Transaction Server and the System Detector objects. The Customer Interface, ATM Control and the Bank Transaction Server receive their respective private keys and UID’s while their public keys and a copy of their UID’s are received by the System Detector which stores them in a hashmap keeping the UID’s as the hashkey and public keys as the hashvalue. As shown in Figure 4.1, once the customer triggers Transfer Funds, message K1.1 Transfer Input is encrypted by the Customer Interface object and is passed on to the System Detector along with the UID of the Customer Interface object of the ATM Client component. On receiving this message the System Detector parses the UID and checks it against the stored UID’s in the hashmap, if a match is found then the Customer Interface’s public key is used to authenticate this message. If there is no match then System Detector knows that a fake object/component is trying to authenticate itself and generates alarm notices. Similarly messages K4.1 and K5.2 are sent to the System Detector by ATM Control object of the
ATM Client component. Message K4.2 is sent to the System Detector by the Bank Transaction Server object of the ATM Server component.

4.4 Centralized System Detector

A system detector can be used in a centralized manner for application systems. All the components interact with one system detector. Objects in components send authenticated notification messages to the centralized system detector that monitors interactions between components. The advantage of a centralized system detector is that it is easy to maintain. All the information about security-relevant interactions among components is contained in a single system detector, which can be modified as the interactions are changing. On the other hand, the centralized system detector can be a bottleneck if the message communications between components are increasing or it does not work as specified.
Figure 4.1 depicts a centralized system detector for the ATM system composed of the ATM Client and Server components. The ATM system detector encapsulates all the interactions between the ATM Client and Server components, which are represented using the state machine. The details of state machine encapsulated in the ATM system detector are described for three use cases for the ATM system in chapter 5 – Withdraw Funds, Query the Balance and Transfer Funds use cases.

Figure 4.3 and Figure 4.4 depicts the centralized System Detector to detect intrusion for Withdraw Funds use case between the ATM Client and the ATM Server component in the ATM system. The ATM Client component notifies the System Detector before sending a withdrawal funds service request. On receiving the withdrawal funds request from the ATM Client component, the ATM Server component notifies the System Detector about this incoming request before processing it. After processing the request the ATM Server component notifies the System Detector and then sends the result to the ATM Client component. On receiving the result from the ATM server component the ATM Client component notifies the System Detector component and further processes the reply and displays the result to the customer. Figure 4.5 depicts the state machine as a statechart in the System Detector object for Withdraw Funds use case (success and failure) in the ATM system.
Figure 4.4 Collaboration diagram for Withdraw Funds use case – Client Side
Figure 4.5 Collaboration diagram for Withdraw Funds use case – Server Side
Figure 4.6 Statechart for Withdraw Funds use case – System Detector

4.5 Distributed System Detector

Distributed system detectors can be used for an application system so that they balance the load across components of the system. A distributed system detector is a set of centralized system detectors working together on different components. A component in a distributed environment needs to communicate with more than one system detectors if it provides or requests services to/from other components. Components send authenticated notification messages to their respective system detectors. Each system detector monitor the communications between components on receiving authenticated notifications from components in a sequential manner as described in the state machine of that system detector. The advantages of a distributed system detector are its capability
to handle requests and ability to scale itself for additional components. The disadvantages are the implementation cost and difficulty in maintenance.

Figure 4.6 shows the collaboration diagram for Send Invoice use case in a Business-to-Business (B2B) electronic commerce system. The Distributed System Detector are Supplier Invoice Detector, Delivery Invoice Detector and Requisition Invoice Detector; these together detect intrusion between the Invoice Agent, Supplier Agent and the Delivery Order Agent components in a B2B ecommerce system. The Supplier Agent notifies the Supplier Invoice Detector that it would be sending an invoice to the Invoice Agent causing the state machine in the Supplier Invoice Detector to transit from Idle to Waiting for Invoice state as shown in figure 4.7. The Supplier Agent sends the invoice to the Invoice Agent. The Invoice Agent notifies the incoming message to the Supplier Invoice Detector causing the state machine to transit from Waiting for Invoice to Processing Invoice. The Invoice Agent notifies the Delivery Invoice Detector that it would be sending the Order Status to the Delivery Order Agent causing the state machine in the Delivery Invoice Detector to transit from Idle state to Preparing Order state as shown in Figure 4.8. The Delivery Order Agent on receiving the Order Status notifies the Delivery Invoice Detector to transit the state machine from Preparing Order to Processing Order state. The Delivery Agent notifies the Delivery Invoice Detector that it would be sending the Order Notification to the Invoice Agent causing the state machine in the Delivery Invoice Detector to transit to Waiting for Order Response state. The Delivery Order Agent notifies the Invoice Agent that the goods have been received. The Invoice Agent notifies this to the Delivery Invoice Detector thus jumping back to the Idle state. The Invoice Agent notifies the Requisition Invoice Detector causing the state machine in the Requisition Invoice Detector to transit from Idle to Preparing Contract state as shown in Figure 4.9. The Requisition Agent that is in the Requisition System notifies the Requisition Invoice Detector causing the Requisition Invoice Detector to jump from Preparing Contract to Processing Contract state. Once the contract is processed the Requisition Agent notifies the Requisition Invoice Detector. On receiving the contract response the Invoice Agent notifies the Requisition Invoice Detector, in case of failure
the state machine transits back to Idle state else it transits to Contract Success state. The Invoice Agent further continues by requesting Funds. The state machine jumps as shown in Figure 4.9. The Invoice Agent further instantiates the payment for this invoice, notifies the Supplier Invoice Detector and returns the result to the Supplier Agent. On receiving the result the Supplier Agent notifies the Supplier Invoice Detector causing the state machine to transit from Waiting for Invoice Response state to Idle state as shown in Figure 4.7.
Figure 4.7 Collaboration diagram for Send Invoice use case
Figure 4.8 State machine for Supplier Invoice Detector
Figure 4.9 State machine for Delivery Invoice Detector
4.6 Implementation and Evaluation the Proposed Approach

This approach is validated through ATM System and E-Commerce (B2B) System case studies. The ATM System includes a secure state-dependent and a secure non-state dependent component and the communication between these two components is monitored by a centralized System Detector, while the E-Commerce System consists of secure non-state dependent components and the communication between these components is monitored by a set of Distributed System Detectors.

To test our approach a hacker thread is created which is initially in a sleep mode. The system runs as it is supposed to. The hacker thread comes out of sleep mode and
initiates/requests services on objects that store sensitive data. The monitor within the component detects these unusual requests as intrusion and generates alarm notices. The hacker thread is also designed to request services from different components and this is caught by appropriate System Detector which generates alarm notices. A malicious object is added to a component such as the ATM Client component, requesting a service from the ATM Server component maliciously. In addition, a malicious object is located outside a component and tested.

The performance overhead of the proposed approach is simulated and analyzed on two case studies. The prototypes are implemented using Java programming language:

ATM System – The total time taken to process is the summation of CPU time (processing the request in the ATM Client and Server), I/O time (time taken to dispense cash, print receipt and eject the card), and User time (time taken by the user to enter his credentials and other details). Table 4.1 depicts the analysis result of the Withdraw Funds use case, Table 4.2 depicts the analysis results of the Transfer funds use case while the analysis results of Query Account are shown in Table 4.3. All the use cases were executed 15 times on the same machine that had AMD Athlon Dual-Core QL-60 1.9 Ghz processor and 2 GB RAM. For simulation purpose the best I/O time is considered, the values taken are 1500 milliseconds to dispense cash, 1000 milliseconds to print receipt and 500 milliseconds to eject card. User time is omitted. The following results were obtained.

\[
\text{Total Time} = \text{CPU Time} + \text{I/O Time} + \text{User Time}
\]

\[
\text{Performance Overhead} = \frac{\text{Security Overhead Time}}{\text{Total Time}} \times 100
\]

Use case: Withdraw Funds
I/O Time = Dispense Cash + Print Receipt + Eject Card
Table 4.1 Performance Analysis of Withdraw Funds use case

| Use case: Transfer Funds | I/O Time = Print Receipt + Eject Card |

<table>
<thead>
<tr>
<th>Messages</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Run 6</th>
<th>Run 7</th>
<th>Run 8</th>
<th>Run 9</th>
<th>Run 10</th>
<th>Run 11</th>
<th>Run 12</th>
<th>Run 13</th>
<th>Run 14</th>
<th>Run 15</th>
<th>Averages</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1.1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.733333333333</td>
</tr>
<tr>
<td>W1.5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.733333333333</td>
</tr>
<tr>
<td>W2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.833333333333</td>
</tr>
<tr>
<td>W2.16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>W2.18</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.933333333333</td>
</tr>
</tbody>
</table>

Security overhead time:
- 5 5 4 3 3 4 6 3 5 5 3 3 6 4 4 4.2

Time without Security:
- 3001 3000 3001 3022 3002 3001 3004 3001 3001 3000 3003 3002 2999 3002 3000 3002.6

Performance Overhead = \( \frac{4.2}{3006.3} \times 100 = 0.139\% \)

Use case: Transfer Funds

I/O Time = Print Receipt + Eject Card
### Table 4.2 Performance Analysis of Transfer Funds use case

<table>
<thead>
<tr>
<th>Messages</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Run 6</th>
<th>Run 7</th>
<th>Run 8</th>
<th>Run 9</th>
<th>Run 10</th>
<th>Run 11</th>
<th>Run 12</th>
<th>Run 13</th>
<th>Run 14</th>
<th>Run 15</th>
<th>Averages</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1.1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.7333333333</td>
</tr>
<tr>
<td>T1.5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.8</td>
</tr>
<tr>
<td>T2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.1066666667</td>
</tr>
<tr>
<td>T3.9,T3A.9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.0933333333</td>
</tr>
<tr>
<td>T3.11,T3A.1</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Security overhead time</th>
<th>5</th>
<th>4</th>
<th>4</th>
<th>5</th>
<th>3</th>
<th>7</th>
<th>4</th>
<th>4</th>
<th>5</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>

**Performance Overhead**

\[
\text{Performance Overhead} = \frac{4.533}{1504.93} \times 100 = 0.301\%
\]

**Use case: Query Account**

**I/O Time = Print Receipt + Eject Card**
Table 4.3 Performance Analysis of Query Account use case

<table>
<thead>
<tr>
<th>Messages</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Run 6</th>
<th>Run 7</th>
<th>Run 8</th>
<th>Run 9</th>
<th>Run 10</th>
<th>Run 11</th>
<th>Run 12</th>
<th>Run 13</th>
<th>Run 14</th>
<th>Run 15</th>
<th>Averages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1.1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.8</td>
</tr>
<tr>
<td>Q1.5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1.0000000000000002</td>
</tr>
<tr>
<td>Q2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.4</td>
</tr>
<tr>
<td>Q2.14</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.9333333333333334</td>
</tr>
<tr>
<td>Q2.16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Security overhead time | 4 | 3 | 3 | 5 | 3 | 4 | 5 | 4 | 4 | 14 | 5 | 5 | 5 | 4 | 4 | 4.8 |

Time without Security

<table>
<thead>
<tr>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1506</td>
</tr>
</tbody>
</table>

Performance Overhead = \( \frac{4.8}{1506.4} \) \times 100 = 0.319%

E-Commerce System – The total time taken to process is the summation of CPU time and User time (time taken by the user to enter processing details). Table 4.4 depicts the analysis result of the Place Requisition use case and Table 4.5 depicts the analysis results of the Process Delivery Order use case. Both the use cases were executed 15 times on the same machine that had AMD Athlon Dual-Core QL-60 1.9 Ghz processor and 2 GB RAM. For simulation purpose User time is omitted. The following results were obtained.

\[
\text{Total Time} = \text{CPU Time} + \text{User Time}
\]
Use case: Place Requisition

Table 4.4 Performance Analysis of Place Requisition use case

<table>
<thead>
<tr>
<th>Messages</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Run 6</th>
<th>Run 7</th>
<th>Run 8</th>
<th>Run 9</th>
<th>Run 10</th>
<th>Run 11</th>
<th>Run 12</th>
<th>Run 13</th>
<th>Run 14</th>
<th>Run 15</th>
<th>Averages</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1.1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td>0.6</td>
</tr>
<tr>
<td>B1.2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1.0666667</td>
</tr>
<tr>
<td>B2.1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0.2666667</td>
</tr>
<tr>
<td>B3.1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>0.1333333</td>
</tr>
<tr>
<td>B9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0.2</td>
</tr>
<tr>
<td>B9.2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0.1333333</td>
</tr>
<tr>
<td>B10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0.2</td>
</tr>
<tr>
<td>B10.2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0.6</td>
</tr>
</tbody>
</table>

Security overhead time

<table>
<thead>
<tr>
<th>Security</th>
<th>Time without Security</th>
<th>Total time</th>
<th>Performance Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>10</td>
<td>10.6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>29.3333333</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td>29.3333333</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td>29.3333333</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td>29.3333333</td>
</tr>
</tbody>
</table>

Performance Overhead = \( \frac{4.4}{15} \times 100 = 29.33\% \)
Use case: Process Delivery Order

Table 4.5 Performance Analysis of Process Delivery Order use case

<table>
<thead>
<tr>
<th>Messages</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Run 6</th>
<th>Run 7</th>
<th>Run 8</th>
<th>Run 9</th>
<th>Run 10</th>
<th>Run 11</th>
<th>Run 12</th>
<th>Run 13</th>
<th>Run 14</th>
<th>Run 15</th>
<th>Averages</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1.1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0.8</td>
</tr>
<tr>
<td>C2.1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.55333333</td>
</tr>
<tr>
<td>C3.1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1.07333333</td>
</tr>
<tr>
<td>C6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1.05333333</td>
</tr>
<tr>
<td>C6.2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.33333333</td>
</tr>
</tbody>
</table>

| Security overhead time | 3 | 2 | 1 | 1 | 5 | 1 | 2 | 1 | 1 | 5 | 11 | 1 | 3 | 2 | 5 | 2.93333333 |
| Time without Security | 2 | 3 | 2 | 2 | 4 | 2 | 3 | 3 | 3 | 3 | 2 | 5 | 2 | 2 | 2 | 2.66666667 |

Performance Overhead = \( \frac{2.933}{5.6} \times 100 = 52.38\% \)

According to the analysis results obtained our approach is well suited for application systems that has few components. The proposed approach can also be applied to application system with many components provided the system is not bound with strict time constraints.
4.7 Summary

This chapter has described the design of secure non-state dependent component in between components. The design has been implemented and verified to validate the design.
CHAPTER 5

CASE STUDY – AUTOMATED TELLER MACHINE SYSTEM

5.1 Introduction

A bank has several Automated Teller Machines (ATMs) which allow a customer to Withdraw funds, Transfer funds and Query Account. Each ATM machine has a user interface, a keyboard, a card reader, a cash dispenser and a receipt printer. Each ATM has several sensitive objects like ATM Card which holds the card number and expiration date, ATM Transaction which holds the customer name, account details and transaction details, Control objects like ATM Control, Device Interface objects like Card Reader Interface and Printer Interface. The detection approach proposed in chapter 4 is validated using this system.

Figure 5.1 shows the architecture diagram of the ATM System with System Detector

![Architecture diagram of a Secure ATM System](image)

This chapter is organized as follows. Section 5.2 describes the PIN Validation use case. Section 5.3 describes the Withdrawal Funds use case. Section 5.4 describes the Transfer Funds use case. Query Account use case is shown in Section 5.5. Section 5.6 shows the consolidated system state machine and Section 5.7 summarizes this chapter.
5.2 PIN Validation Use Case

The ATM Customer can perform Withdraw Funds, Query Account and Transfer Funds transaction types. The PIN Validation use case is common to all those use cases, hence it is an abstract use case.

Figure 5.2 depicts the collaboration diagram for PIN Validation use case at the client side.
When the Card is inserted by the customer the Card Reader Interface notifies this event to the ATM Control object and the client state machine transits from Idle state to Waiting Card Requested state. The customer enters the PIN and the Customer Interface notifies this event to the System Detector, where the system state machine transits from Idle to Preparing PIN Validation Request as shown in Figure 5.4. The Customer Interface further requests the card details from the ATM Card object. The ATM Card object notifies the ATM Control object before sending the card details to the Customer Interface. On this notification the client state machine transits from Waiting Card Requested to Update
ATM Transaction state. The Customer Interface further sends the Customer Info to the ATM Transaction object and the ATM Control object. The ATM Transaction object on receiving the Customer Info immediately notifies the ATM Control object and updated itself, while the client state machine in the ATM Control object transits from Update ATM Transaction to Waiting for PIN state. The ATM Control object notifies the System Detector, to cause the system state machine to transit from Preparing PIN Validation Request state to Receiving PIN Validation Request state and forwards the details to the ATM Server component and waits for the reply.

![Figure 5.4 System Detector’s state machine for PIN Validation use case](image)

Figure 5.5 depicts the collaboration diagram for PIN Validation use case at the server side.
Figure 5.5 Collaboration diagram for PIN Validation use case – Server Side

On receiving the PIN Validation request from the ATM Client component the ATM Server component notifies the System Detector. This causes the system state machine to transit from Receiving PIN Validation Request state to Processing PIN Validation state. The Bank Transaction server then performs confidentiality and integrity checks and notifies the Bank Server Monitor. The PIN Validation Transaction Manager checks the customer entered PIN with the PIN maintained by the system and returns a success or failure result to the Bank Transaction Server object. The entire process has message/events that are notified to the Bank Server Monitor as shown in Figure 5.6.
The Bank Transaction Monitor notifies the System Detector before sending the result to the ATM Client component. This notification causes the system state machine to transit from Processing PIN Validation state to Receiving PIN Validation Result. On receiving the result the ATM Client notifies the System Detector, which causes the system state machine to transit to Idle state as shown in Figure 5.4.

The ATM Control object updates the ATM Transaction which transits the client state machine from Update ATM Transaction state to Waiting for Customer Choice state as shown in Figure 5.3.
5.3 Withdraw Funds Use Case

An ATM Customer can withdraw funds from either his Checking account or Savings account. A detailed description is depicted in this use case.

Figure 5.7 depicts the collaboration diagram for Withdraw funds use case at the client side.

Figure 5.8 depicts the statechart for the ATM Client component and Figure 5.9 depicts the statechart for the communication between the ATM Client and the ATM Server component for Withdraw Funds use case.
Figure 5.8 Client state machine for Withdraw Funds use case

The client state machine begins at Waiting for Customer Choice state. When the customer selects Withdraw Funds the Customer Interface notifies this event to the System Detector, this causes the system state machine to transit from Idle state to Preparing Withdrawal Request state as shown in Figure 5.9. The Customer Interface further continues by requesting the transaction details from the ATM Transaction object. The ATM Transaction object notifies this event to the ATM Control object. The client state machine transits from Waiting for Customer Choice state to Reading ATM Transaction state as shown in Figure 5.8. On receiving the customer details from the ATM Transaction object, the Customer Interface forwards it to the ATM Control object. This causes the client state machine to transit from Reading ATM Transaction state to
Processing Withdrawal state. The ATM Client component forwards this request to the ATM Server component notifying the System Detector and waits for the reply. The system state machine transits from Preparing Withdrawal Request to Receiving Withdrawal Request state as shown in Figure 5.9.

Figure 5.9 System state machine for Withdraw Funds use case

Figure 5.10 depicts the collaboration diagram for Withdraw Funds use case at the server side.
On receiving the Withdrawal Funds request from the ATM Client component the ATM Server component notifies the System Detector. This causes the system state machine to transit from Receiving Withdrawal Request state to Processing Withdrawal state. The Bank Transaction server then performs confidentiality and integrity checks and notifies the Bank Server Monitor. The Withdrawal Transaction Manager checks the customer account for funds and daily debit limit. If all checks are successful then the customer account is debited, Transaction Log is updated and success is returned. Else failure is returned to the Bank Transaction Server object. The entire process has message/events that are notified to the Bank Server Monitor as shown in Figure 5.11.
The Bank Transaction Monitor notifies the System Detector before sending the result to the ATM Client component. This notification causes the system state machine to transit from Processing Withdrawal state to Receiving Withdrawal Result state. On receiving the result the ATM Client notifies the System Detector, which causes the system state machine to transit to Idle state as shown in Figure 5.9.

Figure 5.11 Server state machine for Withdraw Funds use case
The client state machine when notified by incoming messages, accordingly transits to Updating ATM Transaction, Dispensing, Reading ATM Transaction, Printing, Ejecting, Terminating, Idle states as shown in Figure 5.8.

5.4 Transfer Funds Use Case

The ATM Customer can transfer funds from one account to another. The detailed description is shown below.

Figure 5.7 depicts the collaboration diagram for Transfer Funds use case at the client side.

---

**Figure 5.12** Collaboration diagram for Transfer Funds use case – Client Side
Figure 5.13 depicts the statechart for the ATM Client component and Figure 5.14 depicts the statechart for the communication between the ATM Client and the ATM Server component for Transfer Funds use case.

The client state machine begins at Waiting for Customer Choice state. When the customer selects Transfer Funds the Customer Interface notifies this event to the System Detector, this causes the system state machine to transit from Idle state to Preparing Transfer Request state as shown in Figure 5.14. The Customer Interface further continues
by requesting the transaction details from the ATM Transaction object. The ATM Transaction object notifies this event to the ATM Control object. The client state machine transits from Waiting for Customer Choice state to Reading ATM Transaction state as shown in Figure 5.13. On receiving the customer details from the ATM Transaction object, the Customer Interface forwards it to the ATM Control object. This causes the client state machine to transit from Reading ATM Transaction state to Processing Transfer Funds state. The ATM Client component forwards this request to the ATM Server component notifying the System Detector and waits for the reply. The system state machine transits from Preparing Transfer Request to Receiving Transfer Request state as shown in Figure 5.14.

![System state machine for Transfer funds use case](image-url)

**Figure 5.14** System state machine for Transfer funds use case
Figure 5.15 depicts the collaboration diagram for Transfer Funds use case at the server side.

On receiving the Transfer Funds request from the ATM Client component the ATM Server component notifies the System Detector. This causes the system state machine to transit from Receiving Transfer Request state to Processing Transfer state. The Bank Transaction server then performs confidentiality and integrity checks and notifies the Bank Server Monitor. The Transfer Transaction Manager checks the from account for funds. If the check is successful then the from account is debited and the to account is credited with the transfer amount specified by the customer. Transaction Log is updated and success is returned. Else failure is returned to the Bank Transaction Server object.
The entire process has message/events that are notified to the Bank Server Monitor as shown in Figure 5.16.

The Bank Transaction Monitor notifies the System Detector before sending the result to the ATM Client component. This notification causes the system state machine to transit from Processing Transfer state to Receiving Transfer Result state. On receiving the result the ATM Client notifies the System Detector, which causes the system state machine to transit to Idle state as shown in Figure 5.14.

![Figure 5.16 Server state machine for Transfer Funds use case](image)

The client state machine when notified by incoming messages, accordingly transits to Updating ATM Transaction, Reading ATM Transaction, Printing, Ejecting, Terminating, Idle states as shown in Figure 5.13.
5.5 Query Account Use Case

The ATM customer can check his account balance through the Query Account use case. The details are described below.

Figure 5.17 depicts the collaboration diagram for Query Account use case at the client side.

Figure 5.17 Collaboration diagram for Query Account use case – Client Side
Figure 5.18 depicts the statechart for the ATM Client component and Figure 5.19 depicts the statechart for the communication between the ATM Client and the ATM Server component for Query Account use case.
Figure 5.19 System state machine for Query Account use case

The client state machine begins at Waiting for Customer Choice state. When the customer selects Query Account the Customer Interface notifies this event to the System Detector, this causes the system state machine to transit from Idle state to Preparing Query Request state as shown in Figure 5.19. The Customer Interface further continues by requesting the transaction details from the ATM Transaction object. The ATM Transaction object notifies this event to the ATM Control object. The client state machine transits from Waiting for Customer Choice state to Reading ATM Transaction state as shown in Figure 5.18. On receiving the customer details from the ATM Transaction object, the Customer Interface forwards it to the ATM Control object. This causes the client state machine to transit from Reading ATM Transaction state to Processing Query Account state. The ATM Client component forwards this request to the ATM Server component notifying the System Detector and waits for the reply. The system state
machine transits from Preparing Query Request to Receiving Query Request state as shown in Figure 5.19.

Figure 5.20 Collaboration diagram for Query Account use case – Server Side

Figure 5.20 depicts the collaboration diagram for Query Account use case at the server side.

On receiving the Query Account request from the ATM Client component the ATM Server component notifies the System Detector. This causes the system state machine to transit from Receiving Query Request state to Processing Query state. The Bank Transaction server then performs confidentiality and integrity checks and notifies the Bank Server Monitor. The Query Transaction Manager reads the account balance.
Transaction Log is updated and success is returned. The entire process has message/events that are notified to the Bank Server Monitor as shown in Figure 5.21.

The Bank Transaction Monitor notifies the System Detector before sending the result to the ATM Client component. This notification causes the system state machine to transit from Processing Query state to Receiving Query Result state. On receiving the result the ATM Client notifies the System Detector, which causes the system state machine to transit to Idle state as shown in Figure 5.19.

![Figure 5.21 Server state machine for Query Account use case](image)
The client state machine when notified by incoming messages, accordingly transits to Updating ATM Transaction, Reading ATM Transaction, Printing, Ejecting, Terminating, Idle states as shown in Figure 5.18.

5.6 Consolidated System State Machine

Figure 5.22 depicts the System Detector’s state machine which lies in the System Detector component. This state machine begins at the Idle state and takes 4 independent paths, depending on the initial incoming message, which in turn depends on the use case under execution. All paths execute and end back at the Idle state so as to begin a new use case execution.
5.7 Summary

This design of secure state-dependent and secure non-state dependent components using statechart diagrams to detect intrusion within the component and in between components has been validated in this chapter.
CHAPTER 6

CASE STUDY – ELECTRONIC COMMERCE SYSTEM

6.1 Introduction

Electronic commerce or e-commerce consists of buying or selling of products or services over electronic systems such as the internet and other computer networks. Electronic commerce that is conducted between businesses is referred to as business-to-business or B2B. Electronic commerce that is conducted between businesses and consumers, on the other hand, is referred to as business-to-consumer or B2C [13]. In this case study a B2B e-commerce system is used to validate the distributed system detector proposed in Chapter 4.

Figure 6.1 shows the architecture diagram of the E-Commerce System with multiple Detectors.

![Architecture diagram of a E-Commerce System](image)

**Figure 6.1** Architecture diagram of a E-Commerce System
This chapter is organized as follows. Section 6.2 describes Place Requisition use case. Section 6.3 describes Process Delivery Order use case. Section 6.4 describes Confirm Shipment use case. Section 6.5 describes Confirm Delivery use case. Send Invoice use case is shown in Section 6.6. Section 6.7 shows the consolidated system state machine and Section 6.8 summarizes this chapter.

### 6.2 Place Requisition Use Case

Figure 6.2, 6.5 and 6.7 collectively shows the collaboration diagram of Place Requisition use case.

![Collaboration diagram of Place Requisition use case](image-url)
B1: The customer selects items from catalog and requests creation of a requisition.

B1.1: The Customer Interface notifies the Customer Requisition Detector and the state transits from Idle to Preparing Requisition as shown in Figure 6.3.

B1.2: The Customer Interface notifies the Customer Delivery Detector and the state transits from Idle to Preparing Purchase Request as shown in Figure 6.4.

B2: The Customer Interface forwards the request to the Customer Agent.

B2.1: The Customer Agent notifies the Customer Delivery Detector and the state transits from Preparing Requisition to Receiving Requisition Result.
Request as shown in Figure 6.3.

![State machine for Customer Delivery Detector](image)

**Figure 6.4** State machine for Customer Delivery Detector

B3: The Customer Agent instantiates the Requisition Agent, passing to it the customer’s requisition request.

B3.1: The Requisition Agent notifies this incoming request to the Customer Requisition Detector causing the state to transit from Receiving Requisition Request to Processing Requisition as shown in Figure 6.3.

B3.2: The Requisition Agent notifies this incoming request to the Requisition Monitor causing the state to transit from Idle to Wait for Contracts as shown in Figure 6.6.

B4: The Requisition Agent sends a contract query to the Contracts Server.

B4.1: The Contracts Server notifies this to the Requisition Monitor causing the transition of states from Wait for Contracts to Processing Contracts.

B5: The Contracts Server returns the contracts available between the customer and the supplier.

B5.1: The Requisition Agent notifies the Requisition Monitor to change the state from Processing Contracts to Wait for Funds as shown in Figure
6.6.

B6: The Requisition Agent sends a reserve funds request to the Operations Funds Server, to hold the funds from a given contract for this requisition.

B6.1: The Operations Funds Server notifies this to the Requisition Monitor causing the transition of states from Wait for Funds to Processing Funds.

B7: The Operations Funds Server confirms that the funds have been reserved.

B7.1: The Requisition Agent notifies the Requisition Monitor to change the state from Processing Funds to Wait for Store as shown in Figure 6.6.

Figure 6.5 Collaboration diagram of the Requisition System

B8: The Requisition Agent approves the requisition and sends it to be stored at the Requisition Server.
B8.1: The Requisition Server notifies this to the Requisition Monitor causing the transition of states from Wait for Store to Idle.

B9: The Requisition Agent notifies the Customer Requisition Detector causing the state to transit from Processing Requisition to Receiving Requisition Result.

B9.1: The Requisition Agent sends the requisition status to the Customer Agent.

B9.2: The Customer Agent notifies the Customer Requisition Detector causing the state machine to transit to Idle state as shown in Figure 6.3.

B10: The Customer Agent notifies the Customer Delivery Detector to transit the state machine from Preparing Purchase Request to Processing Purchase Request.

B10.1: The Customer Agent forwards the purchase request to the Delivery Order Agent.

B10.2: The Delivery Order Agent notifies the Delivery Monitor about this
incoming message. This causes the state machine of the Delivery monitor to transit from Idle state to Wait for Contracts as shown in Figure 6.8.

B11: The Delivery Order Agent creates a new delivery order and stores it with the Orders Server.

B11.1: The Orders Server notifies this to the Delivery Monitor causing the state to transit from Wait for Contracts to Idle.

B12: The Customer Agent sends the requisition status to the Customer Interface

B13: The Customer Interface displays the requisition status to the customer.

Figure 6.7 Collaboration diagram of the delivery System

Figure 6.8 State machine for the Delivery Monitor in the Delivery System
6.3 Process Delivery Order Use Case

Figure 6.9 shows the collaboration diagram for Process Delivery Order use case

![Collaboration diagram for Process Delivery order use case](image)

**Figure 6.9** Collaboration diagram for Process Delivery order use case

C1: The supplier requests a new delivery order.

B1.1: The Supplier Interface notifies the Supplier Delivery Detector and the state machine transits from Idle to Order Request Received state as shown in Figure 6.10.

C2: The Supplier Interface forwards the request to the Supplier Agent.

C2.1: The Supplier Agent notifies the Supplier Delivery Detector and the state machine transits from Order Request Received state to Preparing Order Request state.

C3: The Supplier Agent sends the order request to the Delivery Order Agent.
C3.1: The Delivery Order Agent notifies the Supplier Delivery Detector and the state machine transits from Preparing Order Request state to Processing Order.

C4: The Delivery Order Agent queries the Orders Server.

C5: The Orders Server sends back a delivery order.

C6: The Delivery Order Agent notifies the Supplier Delivery Detector and the state machine transits from Processing Order to Preparing Inventory Request.

   C6.1: The Delivery Order Agent sends the delivery order to the Supplier Agent.

   C6.2: The Supplier Agent notifies the Supplier Delivery Detector and the state machine transits from Preparing Order Request state to Idle state.

C7: Supplier Agent checks the items availability.

C8: Availability is confirmed.

C9: The Supplier Agent sends the order status to the Supplier Interface.

C10: The Supplier Interface displays the delivery order and inventory information to the supplier.
Figure 6.10 State machine for Supplier Delivery Detector

6.4 Confirm Shipment Use Case

Figure 6.11 depicts the collaboration diagram for Confirm Shipment use case
Figure 6.11 Collaboration diagram for Confirm Shipment use case

S1: The supplier inputs the shipping information.

S1.1: The Supplier Interface notifies the Supplier Delivery Detector and the state machine transits from Idle to Waiting for Supply Request state as shown in Figure 6.12.

S1.2: The Supplier Interface also notifies the Customer Delivery Detector and the state machine transits from Idle to Waiting for Order in Delivery Agent state as shown in figure 6.13.

S2: The Supplier Interface forwards the request to the Supplier Agent.

S2.1: The Supplier Agent notifies the Supplier Delivery Detector and the state machine transits from Waiting for Supply Request state to Preparing Order state.
S3: The Supplier Agent updates the inventory stored at the Inventory Server.

S4: The Supplier Agent notifies the Supplier Delivery Detector and the state machine transits from Preparing Order state to Waiting for Order state.
   S4.1: The Supplier Agent sends the order status to the Delivery Order Agent.
   S4.2: The Delivery Order Agent notifies the Supplier Delivery Detector and the state machine transits from Waiting for Order state to Idle state.
   S4.3: The Delivery Order Agent notifies the Customer Delivery Detector and the state machine transits from Waiting for Order in Delivery Agent state to Preparing Order state as shown in figure 6.13.
Figure 6.13 State machine for Customer Delivery Detector

S5: The Delivery Order Agent updates the Orders Server.

S6: The Delivery Order Agent notifies the Customer Delivery Detector and the state machine transits from Preparing Order state to Waiting for Order in Customer Agent state.

S6.1: The Delivery Order Agent sends the order status to the Customer Agent.

S6.2: The Customer Agent notifies the Supplier Delivery Detector and the state machine transits from Waiting for Order in Customer Agent to Idle state.

S7: The Customer Agent forwards the order status to the Customer Interface.

S8: The Customer Interface displays the order status to the customer.
6.5 Confirm Delivery Use Case

Figure 6.14 depicts the collaboration diagram for Confirm Delivery use case.

**Figure 6.14** Collaboration diagram for Confirm Delivery use case

R1: The customer acknowledges the receipt of shipment.

R1.1: The Customer Interface notifies the Customer Requisition Detector and the state machine transits from Idle to Preparing Shipment Request state as shown in Figure 6.15.

R1.2: The Customer Interface also notifies the Customer Delivery Detector and the state machine transits from Idle to Waiting for Customer Confirmation state as shown in figure 6.16.
Figure 6.15 State machine for Customer Requisition Detector

R2: The Customer Interface send the customer information to the Customer Agent.

R2.1: The Customer Interface notifies the Customer Delivery Detector and the state machine transits from Waiting for Customer Confirmation to Preparing Shipment Request state.

R3: The Customer Interface notifies the Customer Delivery Detector that it would be sending the shipment details to the Delivery Order Agent. This causes the state machine of the Customer Delivery Detector to transit from Preparing Shipment Request to Waiting for Shipment state.

R3.1: The Customer Agent sends the Shipment Details message to the Delivery Order Agent.

R3.2: The Delivery Order Agent notifies this incoming message to the Customer Delivery Detector to transit the state machine from Waiting for Shipment state to Idle state as shown in Figure 6.16.
Figure 6.16 State machine for Customer Delivery Detector

R4: The Delivery Order Agent updates the status at the Orders Server.

R5: The Customer Agent notifies the Customer Requisition Detector that it would be sending the shipment details to the Requisition Agent. This causes the state machine of the Customer Requisition Detector to transit from Preparing Shipment Request to Waiting for Shipment state as shown in Figure 6.15.

R5.1: The Customer Agent sends the Shipment Details message to the Requisition Agent.

R6: The Requisition Agent notifies this incoming message to the Customer Requisition Detector causing the state machine to transit to Idle state.

R6.1: The Requisition Agent updates the status of the requisition stored at the Requisition Server.

R7: The Requisition Agent commits the funds for this requisition with the Operations
Funds Server.

R8: The Operation Funds Server replies back with successful commitment.

6.6 Send Invoice Use Case

D1: The Supplier Agent notifies the Supplier Invoice Detector that it would be sending an invoice to the Invoice Agent causing the state machine in the Supplier Invoice Detector to transit from Idle to Waiting for Invoice state as shown in Figure 6.18.

D1.1: The Supplier Agent sends the invoice to the Invoice Agent as depicted in Figure 6.17.

D1.2: The Invoice Agent notifies this incoming message to the Supplier Invoice Detector causing the state machine to transit.

Figure 6.17 Collaboration diagram for Send Invoice use case

79
D2: The Invoice Agent notifies the Delivery Invoice Detector that it would be sending the Order Status to the Delivery Order Agent causing the state machine in the Delivery Invoice Detector to transit as shown in figure 6.19.

D2.1: The Invoice Agent subscribes to the Delivery Order Agent.

D1.2: The Delivery Order Agent notifies this incoming message to the Delivery Invoice Detector causing the state machine to transit.

![State machine for Supplier Invoice Detector](image)

**Figure 6.18** State machine for Supplier Invoice Detector
Figure 6.19 State machine for Delivery Invoice Detector

D3: The Delivery Agent notifies the Delivery Invoice Detector that it would be sending the Order Notification to the Invoice Agent causing the state machine in the Delivery Invoice Detector to transit.

D3.1: The Delivery Order Agent notifies the Invoice Agent that the goods have been received.

D3.2: The Invoice Agent notifies this to the Delivery Invoice Detector.
D4: The Invoice Agent notifies the Requisition Invoice Detector causing the state machine in the Requisition Invoice Detector to transit from Idle to Preparing Contract state as shown in Figure 6.20.

D4.1: The Invoice Agent sends a Contract Query to the Requisition System.
D4.2: The Requisition Agent notifies the Requisition Invoice Detector.
D4.3: The Requisition Agent notifies the Requisition Monitor as shown in Figure 6.22 the state machine transits from Idle state to Preparing Contract state.
Contract state.

Figure 6.21 Collaboration diagram of the Requisition System

D5: The Requisition Agent sends a contract query to the Contracts Server.

B5.1: The Contracts Server notifies this to the Requisition Monitor causing the transition of states from Preparing Contracts to Processing Contracts.

D6: The Contracts Server returns the contracts available between the customer and the supplier.

D6.1: The Requisition Agent notifies the Requisition Monitor.

D6.2: The Requisition Agent notifies the Requisition Invoice Detector.

D6.3: The Requisition Agent confirms the contract.

D7: The Invoice Agent notifies the Requisition Invoice Detector causing the state machine in the Requisition Invoice Detector to transit from Contracts Success to
Preparing Funds state as shown in Figure 6.20.

D7.1: The Invoice Agent sends a Funds Query to the Requisition System.
D7.2: The Requisition Agent notifies the Requisition Invoice Detector.
D7.3: The Requisition Agent notifies the Requisition Monitor.

![State machine for Requisition Monitor](image)

Figure 6.22 State machine for Requisition Monitor

D8: The Requisition Agent sends a reserve funds request to the Operations Funds Server, to hold the funds from a given contract for this requisition.

B8.1: The Operations Funds Server notifies this to the Requisition Monitor causing the transition of states from Preparing Operations Funds to Processing Operation Funds.

D9: The Operations Funds Server confirms that the funds have been reserved.

D9.1: The Requisition Agent notifies the Requisition Monitor.
D9.2: The Requisition Agent notifies the Requisition Invoice Detector.
D9.3: The Requisition Agent confirms the funds.
D10: The Invoice Agent sends the invoice to the Accounts Payable Server.

D11: The Accounts Payable Server sends the payment status to the Invoice Agent.

D12: The Invoice Agent stores the invoice at the Invoice Server.

D13: The Invoice Agent sends the electronic payment to the customer’s bank via the Bank Server Interface.

D14: Bank Server Interface sends the electronic funds to the customer’s bank for payment to the supplier.

D15: The Invoice Agent notifies the Supplier Invoice Detector.
   
   D15.1: The Invoice Agent sends the invoice status to the Supplier Agent.
   
   D15.2: The Supplier Agent notifies the Supplier Invoice Detector to transit the state machine from Waiting for Invoice Response state to Idle state.

D16: The Supplier Agent sends the invoice status to the Supplier Interface.

D17: The Supplier Interface displays the invoice status to the Supplier.
6.7 Consolidates System State Machine

The e-commerce case study has lot of system detectors and monitors. The following diagrams depict their consolidated state machines for all the use cases taken together.

![Diagram of Consolidated state machine for Requisition Monitor]

**Figure 6.23** Consolidated state machine for Requisition Monitor
Figure 6.24 Consolidated state machine for Customer Requisition Detector
6.8 Summary

This design of secure non-state dependent components using statechart diagrams to detect intrusion within a component and in between components has been validated in this chapter.
CHAPTER 7
CONCLUSIONS AND FUTURE WORK

This thesis addresses an approach to detect malicious software engineer intrusion detection between components. We have proposed an approach that uses statechart model to detect illegal application flow between components. The System detector monitors the information flow between components and generates alarms whenever the flow deviates from the actual behavior as defined in the statechart model of the secure component.

This thesis develops the system detectors, which encapsulate the state information of interactions between components. The system detectors can be used as Centralized or Distributed System Detectors. In general, Centralized System Detectors are used when we have fewer components in an application while Distributed System Detectors are used when the components are more. Sender authentication mechanism is developed to make sure that the System Detector does not transit states on receiving messages from fake components. Finally, to validate our approach, secure software components are designed and implemented on the ATM System and E-Commerce System case studies.

This research can be connected to some future work. The current approach generates alarms but doesn’t prevent the system from serving the malicious call. Future work can be done to prevent a component from serving a malicious service request. We are currently using the statechart model to define the flow of an application and check for intrusions. Instead of statechart model, model checking can be used to achieve malicious software intrusion detection in a different way.
REFERENCES


