MONITORING COMPONENTS BY USING ASPECTS AND CONTRACTS IN WRAPPERS

by

(MICHAEL) XIAOFENG YANG

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
January 2011

Copyright © (Michael) Xiaofeng Yang, 2011
Abstract

The re-usability and modularity of components reduce the cost and complexity of the software design. It is difficult to predict run-time scenarios covering all possible circumstances to ensure that the components are full compatible with the system. Monitoring run-time behaviours of components presents a close view of the component qualities. The existing monitoring approaches either implement applications with built-in monitoring features, or observe the external resource and events to predict the status of the components. In this thesis work, we propose an approach to monitor the run-time behaviours of components with aspect-oriented wrappers and contracts.

We design monitoring wrappers to encapsulate the monitored components. The wrapper has the access to interfaces and properties of the wrapped component. We adopt the methodology of Design by Contract to enforce security policies on component wrappers. The contracts define the mutual obligations of two interacting components. The policies implemented in contracts are woven into component wrappers as separate aspect modules. If the component contains any flaws, the wrappers can monitor the behaviours and prevent failures propagating into the wrapped components and the rest of the system. This approach assures that the system is running in a safe environment with the erroneous behaviours or failures detected appropriately. Secure access between the wrappers guarantees a secure environment for the
wrapped components.

We conducted experiments on the run-time monitoring of SQL Injection and Cross Site Scripting attacks. We designed cross-cutting concerns such as logging for components to illustrate monitoring components without touching the underlying components. Monitoring on access control is also possible and feasible to add as an additional concern and is also demonstrated in the experiments. The results show that the framework is very flexible to impose separate policies as aspects on component wrappers without the modifications of the underlying components.
Acknowledgments

I would like to express my sincere appreciation to my supervisor, Dr. Mohammad Zulkernine, for his endless assistance, guidance, and advice. I am very grateful to him for being a patient professor.

I would like to thank my group members at Queen’s Reliable Software Technology (QRST) group. I very much appreciated the discussions and the valuable thoughtful suggestions and advice from all my colleagues and especially from Atef Mohamed, Umair A. Khan and Hsiao-Ming Tsou, whom I thank for your encouragement and support.

I like to express my sincere gratefulness to my girlfriend Zhipei Qin, for her understandings, endless supports and encouragements during my program.

I dedicate this work to my beloved parents for their understanding, encouragements and their support during the time when I was struggling with this work.

Thank you to all my friends who encouraged and supported me!
# Table of Contents

Abstract

Acknowledgments

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Motivation

1.2 Problems

1.3 Overview

1.4 Contributions

1.5 Organization

Chapter 2: Background and Related Work

2.1 Background

2.1.1 Execution Monitoring

2.1.2 Component-Based Software System

2.1.3 Component Wrappers

2.1.4 Aspect-Oriented Programming

2.1.5 Design by Contracts

2.2 Related Work

2.2.1 Security for Component Software

2.2.2 Protective Wrappers for Security

2.2.3 Aspect-Oriented Development for Security

2.2.4 Security Contracts

2.2.5 Security Monitoring

2.3 Summary
Chapter 3: Wrappers for Component Monitoring

3.1 Monitoring Overview

3.2 Enforce Additional Policies into Wrappers with Aspects

3.3 Design and Deployment
    3.3.1 Overview
    3.3.2 Design and Implementation
    3.3.3 Deployment of Component Wrapper

3.4 An Example Application
    3.4.1 Implement the Wrapper
    3.4.2 Implement Monitoring Code
    3.4.3 Deploy the Wrapper

3.5 Monitoring Process of Wrapper
    3.5.1 Run-time Verification of Behaviours in Wrappers
    3.5.2 Event-based Monitoring by Wrappers

3.6 Summary

Chapter 4: Security Contracts in Aspects

4.1 Security by Contracts and Aspects

4.2 Enforcing Security Contracts into Component Wrappers
    4.2.1 Overview of Security Contracts
    4.2.2 Process of Enforcing Contracts into Wrappers

4.3 Aspect Oriented Security Contracts
    4.3.1 Architecture of Security Contracts in Aspects
    4.3.2 Design of Security Contracts in Aspects

4.4 Integrating Contracts into Wrappers by Aspects

4.5 Summary

Chapter 5: Experimental Evaluation

5.1 Evaluation Environment
    5.1.1 Target Platform
    5.1.2 Target Vulnerabilities
    5.1.3 Target Example Application
    5.1.4 Supporting Tools

5.2 Experiments
    5.2.1 SQL Injection Monitoring
    5.2.2 Cross Site Scripting Monitoring
    5.2.3 Role Based Access Control Monitoring

5.3 Summary

Chapter 6: Conclusion and Future Work

6.1 Conclusion
6.2 Limitations and Future Work ............................................ 124

Bibliography ................................................................. 126
List of Tables

2.1 Comparisons between Different Monitoring Mechanism ..... 17
2.2 Security Monitoring Related Work .......................... 36
# List of Figures

1.1 Security Problems in Component Communications .......................... 4
1.2 Software Life Cycle and the Monitor Development Framework .......... 7
1.3 Working Environment of Component Monitor ............................... 9

2.1 A Simple Monitor System .................................................... 13
2.2 Component-Based Software System ....................................... 18

3.1 Anatomy of the Monitoring Approach ................................. 45
3.2 Component Wrapper .......................................................... 52
3.3 Process of Method Call in the Component Wrapper ...................... 54
3.4 Sequence Diagram of Wrapper Deployment ............................. 56
3.5 Deployment Aspect for Component Wrappers ........................... 57
3.6 A GUI Application using TemperatureConverter service ............... 58
3.7 Incorporating TemperatureConverter Component without Wrappers .. 59
3.8 Wrapper for TemperatureConverter Component .......................... 60
3.9 Logging Policy for Component Wrapper .................................. 61
3.10 Enforce Validation Policy for Component Wrapper .................... 62
3.11 The Aspect for Instrumenting into the new Wrapper .................. 64
3.12 Wrapper Monitoring Process ................................................ 66
3.13 Process Flow of Evaluating Intercepted Method Calls .................. 69

4.1 Sequence Diagram of Using Contracts for Component Monitoring ..... 73
4.2 Contracts Constraints between FarenheitGUI and TemperatureGUI components ................................................................. 75
4.3 Log Information with Contracts Enforced ................................ 76
4.4 Comparison of Different Mechanisms of Instrumentation .............. 77
4.5 Example of Enforcing Behaviour Contract in Wrappers ................ 78
4.6 Login Contract for Component Wrapper ................................... 79
4.7 Authorization Contract for Component Wrapper ........................ 80
4.8 Applying Contracts as Aspects into Component Wrappers .......... 82
4.9 Weaving Security Contracts into Component Wrappers ............... 84
4.10 Enforcing Security Behaviour Contract with AOP ..................... 85
4.11 Monitoring Framework Integration with Wrapper and Security Contracts with Aspects ........................................ 87

5.1 MVC Design Pattern in WebGoat Architecture ......................... 94
5.2 SQLInjection Component in WebGoat Application ..................... 95
5.3 Sequence Diagram of Executing SQL Injection Actions .................. 98
5.4 Scenario of SQL Injection in Login Action Component .................. 99
5.5 Interfaces of Login Component ........................................... 101
5.6 Wrapper for Login Component ............................................. 103
5.7 OCL specification for SQL Injection ....................................... 104
5.8 Security Contracts of Login SQL Injection ............................... 105
5.9 Security Aspect Code of Login SQL Injection ............................ 107
5.10 Process Flow of Cross Site Scripting Problem ......................... 108
5.11 Cross Site Scripting Code .................................................. 109
5.12 Web Page Generated by Cross Site Scripting ............................ 110
5.13 Wrapper Component for XSS Component ............................... 112
5.14 OCL Constraints for XSS Component .................................... 113
5.15 Security Contracts for XSS Component .................................. 114
5.16 Security Aspects for XSS Component .................................... 115
5.17 Adding Additional Access Control on Interface .......................... 117
5.18 Wrapper Component for RBAC Component .............................. 118
5.19 OCL Constraints for RBAC Component .................................. 119
5.20 Security Contracts for RBAC Component ............................... 120
5.21 Security Aspects for RBAC Component .................................. 121
Chapter 1

Introduction

1.1 Motivation

A component is a piece of executable software module with published interfaces. Building large-scale and complex software systems from available reusable components can achieve three goals: increasing return on investment, decreasing time to market, and assuring quality and reliability [73 78]. The components that are compliant to specific functionality requirements are selected for the composition of the system. Components are separately implemented by different developers from external market sources, for example, COTS (Commercial Off-The-Shelf) [83 88], or can be an already developed or existing in-house software parts with the equivalent functionality. Component-based software engineering aims at optimizing software component re-usability and at developing infrastructures for open and rapidly evolving applications such as large-scale enterprise software and embedded systems. The separately developed components can easily be assembled and added to, and the old ones can be replaced easily [78]. The individual modular components can be acquired
from different sources and are designed with different security requirements, different programming languages, or even from different framework models, such as EJB [17], CORBA [3, 71], OSGi [19], open sourced Fractal [26], and Spring [79]. The quality of components from different environments can vary in great degree from each other. In this way, the security impact on the whole system has a high dependency on all individual components, not on a single one. Components that are secure in one environment may not be secure in a different environment. With the growing size and complexity of applications, support for quality monitoring becomes essential.

To achieve re-usability, a component is designed with a specified interface. The interfaces required for one system may not be necessary for the other one, thus exposing potential for malicious use. To achieve modularity, the components are designed with only functional concerns, leaving non-functional features outside, such as logging, security checking, and monitoring. However, such non-functional concerns are necessary for secure communication among components.

It is challenging to guarantee the component’s quality during the development phase because of incompatibilities between different component models. Monitoring run-time behaviours is both necessary and essential to observe the executions and to discover the security risks in real scenarios. The security of a single candidate component cannot guarantee the security of the enclosing application. Although the quality of the components can be evaluated carefully before integrating into the application system, the run-time risks still occur during execution when working with the rest of the components developed with different quality criteria. The behaviours of the composite system should take all individual components into consideration for quality control. All participating components are primarily responsible for their
own provided functionalities. When the size and complexity of the system grow, even though the component itself is reliable, errors could arise by the incompatible interactions with the rest of the system, for example, unchecked data input and unhandled exceptions. The unexpected business logic errors should also be considered from the system-level view to assure a secure environment for the system.

1.2 Problems

Although a component-based system is adopted for reducing the cost of system implementation, the mismatches between acquired components and the system with perspective of security concerns are causing the inconsistency of system behaviours. The component is generally known only by its provided interfaces. The inner implementations are hidden from requesting clients. As the internals of the components are hidden from users, the components are still prone to carrying erroneous and malicious codes. Thus, the major security issues are raised: Confidentiality, Integrity and Availability.

Confidentiality indicates that a security model is desired to prevent the unauthorized disclosure of information, such as the data flow from a secured component to a malicious component. For example, as Figure 1.1 shows, a Client component accesses the service of a Server component via an Authentication component. The requests from Client are redirected by the Authentication component and forwarded to Server component for service process. The Authentication is a reusable component performing the authentication business with a remote UserDirectory component. If Authentication component exposes too much context information, as not expected by the system, the Client could retrieve resource information and to interact directly
with the Server component bypassing the authentication, thus causing the information leakage from Server in an unwarranted manner.

![Security Problems in Component Communications](image)

**Figure 1.1: Security Problems in Component Communications**

**Integrity** specifies a component’s compliance to system requirements. A component is developed mostly by a third party under different security criteria. Although the functionalities of the released components meet the system requirements, they may still carry a malicious code making improper use of system resource or information assets, which violates the security policies of the composite system. In the example mentioned above, the Authentication component receives the user information from a request sent by the Client component, and authenticate the user request by verifying with the UserDirectory. The system requires that input validation should be taken into consideration to prevent security issues such as SQL Injection and XSS (Cross Site Scripting). However, to make a component reusable in a wide variety of contexts, with the purpose of maximum re-usability, the component developer may
only consider the domain-specific security features, assuming that the data received has been sanitized. The different understandings of security requirements between the reusable components and system architecture may introduce the primary security breaches.

**Availability** determines how a component service should be accessed. Only the services required by the system should be open and exposed. **Hidden Interface** is a unique issue to the component-based system. It is a way for malicious users to exploit exposed context and use the context to further retrieve information assets and system resources. A component is reusable by the system developer without designing the component unit from scratch. To make the component reusable in different systems with maximum degrees, developers expose as many interfaces as they provide for components so that users can access most of the functionalities of the components. However, the available service should be customizable to adapt to particular usage of different systems. The unused ones are protected and the non-existing ones can be added for a particular purpose. The **Hidden Interface** issue can be resolved by adding a component wrapper as we use in our work. Even though the interfaces are publicly available, but are assessed by the wrapper when they are accessed.

Therefore, even though the components have gone through a quality control process during design and implementation, the behaviours of the component may vary in great degrees when incorporated in different systems under different security policies. The errors can still arise at run-time, caused by undetected problems that were not possible to discover under the component vendor’s environment, such as logical errors, unavailable resources on which it depends, and incorrect inputs from other interacting
components. When the components are selected or released, the required functionality or expected behaviours such as model and specifications should be described in clear documents. However, in most situations, the documents of the acquired components are poor, lacking many details of their internal structures. The system designer needs to know the pre-conditions and post-conditions regarding the robustness of the components that are either procured from the market or already available within the internal organization. We use security contracts in our work to enforce policies on the acquired components via wrappers separately from the components.

1.3 Overview

In our thesis work, we present an approach of monitoring some run-time behaviours of components using aspects and contracts in wrappers. Component wrappers are used to isolate the monitored components from the rest of the system. Wrappers are designed to protect and observe transactions between wrapped components and the system. Wrappers consist of encapsulation and monitoring functionality. We instrumented monitoring features into wrappers with aspect-oriented methodology. Aspect-oriented technology has been used for software security, by using bytecode instrumentation during class loading and recompilation [77] of source code [42] [34]. As the source code of the acquired components are usually not available, we propose the approach of instrumenting security monitoring code into the wrappers, instead of the underlying components. The monitoring features are added by aspects invoked from wrappers. The aspects define where and when the wrappers should evaluate the incoming requests and what events should be captured. Different from traditional ways to define aspects, we design the advice module separately in security contracts.
The advice defines how to evaluate the captured events and what to do against the monitored events.

Design by contract is a paradigm initiated by Bertrand Meyer [59]. The contract defines the obligations the object should achieve to ensure the safety of the objects in the environment. The obligations include pre-conditions, post-conditions, and invariants, which specify the required contracts in different stages. As the component is a software module with only interfaces public, it is applicable to adopt these conditions and it is also the best fit to verify the safety of components against the interfaces.

![Software Life Cycle and the Monitor Development Framework](image)

Figure 1.2: Software Life Cycle and the Monitor Development Framework

The proposed monitoring development framework is divided into different stages as shown in Figure 1.2 from the perspective of Software Development Life Cycle (SDLC).

In the Requirement Specification stage, the security policies are specified as the enforceable contracts against method calls or access of properties on the target components. The policy consist of three essential parts to enforce it on the component:
the context of the monitoring, which is the monitored component; the security contract, which is an implementation for run-time verification of the received context information such as request data; the aspects which define the position to enforce the policy and execute the security contract. The aspect code is instrumented into the component wrappers with the security contract module in the Design & Implementation stage.

In the Design & Implementation stage, the component wrapper is implemented and associated with the underlying component using adapter pattern and proxy patterns [39]. They are used to adapt the component into the system with a compatible interface and to create a proxy instance on the underlying component, which is the wrapper in our work. The wrapper enhances the component’s functions while keeping the original ones untouched. It also contains the reference to the component, so that it can delegate requests to the underlying components directly without delay. The security contracts are implemented separately and specified as the reference in the aspects indicated in the security policies.

After the security contracts and the component wrapper are created, the aspects are required to configure the wrappers and contracts to form the composite monitoring system in the Deployment stage. The aspects for the wrapper is used to deploy the wrapper into the existing system. It defines the places in the system where the components need to be monitored. After the wrapper is deployed, the requests to the underlying component are dispatched to the associated component wrapper. The aspects for the contracts are used to instrument for run-time verification as defined by the security contracts into the wrappers, keeping the underlying component untouched. The aspects instrumented into the wrapper invoke the security contracts
When the wrapper intercepts the desired security events.

By the instrumentation of the wrapper and its security contracts, the deployed wrapper communicates on behalf of its wrapped component. The security contract, component wrapper, and aspect are three separate modules that compose the deployed system with monitoring features. Each module is easy to add, remove or replace, increasing the flexibility and the extensibility of the framework.

![Diagram: Working Environment of Component Monitor](image)

Figure 1.3: Working Environment of Component Monitor

The overview of the deployed system with the wrapper, aspect, and contract is shown in Figure 1.3. The *Component Container* is the environment to deploy the component. The wrapper is deployed with aspects into the *Component Container* associating with the wrapped component. The security contracts are instrumented into the wrappers via the monitoring aspects that include the join points to weave. The wrapper monitors the events between the wrapped component and the rest of the system. When the captured event (method call in our work) violates the security policies implemented in the security contracts, the wrapper stops the execution, logs the information and reports the security violations.
1.4 Contributions

Our framework does not rely on the application server to implement security concerns, as different application servers have different mechanisms to implement security management. Many popular components are running on vendor-based application servers. The application server offers services, such as transaction management, messaging, mailing, and directory interface. Unfortunately, there exist a few problems with these standards. First of all, the usage of these standards is too complex. Writing a component requires a set of XML files, home interfaces, and remote/local interfaces. Even worse, almost half of the deployment descriptors were vendor specific, migrating an application from vendor A to B was not transparent any more [62]. Our framework is targeted to Plain Old Java Object (POJO), which is a regular Java object. No customized annotations are required to write the component associated with any particular application server. Therefore, our approach is applicable to universal component models. We believe that this approach can be used to observe the run-time security behaviours, while being able to integrate various components from different vendors.

To achieve these contributions, we have done the following tasks:

- Propose a monitoring development framework by using aspects and contracts in wrappers.

- To minimize the instrumentation overhead into the existing components, we design the component wrappers with run-time monitoring features using adapter and proxy patterns.
• Demonstrate the architecture design of enforcing contracts with AOP (Aspect-Oriented Programming) into wrappers.

• Illustrate how the aspects are weaved into wrappers to monitor underlying components.

• Implement the framework and conduct experiments for monitoring popular security risks in a web-based vulnerable JavaEE application, WebGoat.

1.5 Organization

The rest of the thesis is organized as follows:

Chapter 2 introduces the background on component-based software system, its related security problems, and the current monitoring approaches for these security problems.

Chapter 3 introduces the approach of monitoring on components with wrappers.

Chapter 4 illustrates the implementation of security contracts with aspect-oriented programming and the integration with wrappers.

Chapter 5 presents the implementations and experiments of the monitoring framework.

Chapter 6 summarizes our thesis work and outlines future work.
Chapter 2

Background and Related Work

In this chapter, we introduce the background knowledge in Section 2.1. Section 2.1.1 presents the overview of execution monitors by showing the essential parts of the monitoring systems. Section 2.1.2 presents the background knowledge about component-based software systems. Section 2.1.3 describes protective wrappers for component-based system. Section 2.1.4 and Section 2.1.5 introduces aspect-oriented programming technology and contracts. Section 2.2 presents the existing work related to our research. Section 2.3 concludes this chapter.

2.1 Background

2.1.1 Execution Monitoring

Software monitoring is a mechanism to observe the run-time behaviours of software applications. Such monitoring is an ideal method to determine whether the software complies with its expected behaviour. It allows the system to analyze and recover from
detected errors and faults, providing additional defences against malicious attacks.

A sensor is placed into an entity monitoring its state changes as illustrated in Figure 2.1. The entity is the unit in the target system which is observed by the monitoring system, such as the memory space state, which are variables of the target system. The events such as request events or the state changes on the entity are observed by the monitor system. By interpreting the events, the monitoring system evaluates the behaviours according to the defined security policies, for example, the number of requests should not exceed a pre-determined number [69]. The monitoring system also responds to erroneous events to protect the target system.

Figure 2.1: A Simple Monitor System

Monitoring technology can be used for many other purposes. Originally it was used with software debugging, testing, validation, performance evaluation, and system maintenance in a complimentary manner [40, 21, 37, 31, 80]. It has been used for profiling, optimization, and analysis of complex systems, such as failure detection and self-recovery for large-scale systems, in which the debugging is becoming difficult [35, 45, 49]. Monitoring technology has also been used in the component-based software system life cycle to track the state of the software system: unit testing during
the development phase of individual components, the integration testing in the deployment, and the performance monitoring in the maintenance stage. The monitoring is to assure that the run-time behaviours of the system does not violate the system requirements \cite{22}. Functional correctness can be validated by analyzing the data exchanged among services. Although testing can be used to validate the functional correctness, the testing results cannot foresee all those changes that may happen at run-time. Thus we need to shift the validation from testing efforts to run-time and introduce the idea of continuous monitoring, which is the task of the monitoring framework.

The monitor gathers information from a computational process as it executes and can be classified by its functionality \cite{69}: dependability, performance enhancement, correctness checking, and security. The monitor is considered as an external observer of the target system. The observer collects the information of the system while it is running and it also reacts in a timely manner to events occurring in the system, logging the events, blocking the actions, and restoring the component to original states. Correctness checking is to ensure that an application is consistent with the system specifications, while security checking is to assure the application is not breaking the system’s security rules.

Security monitoring detects security violations such as the abuse of system resource and the unauthorized access. The monitoring system taps a sensor to the target system to listen to the events occurred on the target system. The sensor is an event listener that observes the behaviour of a specific entity of the target system. It listens to the internal changes on the observed entity, when the registered event of the entity occurs, the sensor generates a notification to trace the changes on the
entity. Tracing is performed synchronously with the change in the value of the entity [69]. The sensors are usually placed in the target system where the state changes of the observed entity occur. The sensor contains the monitoring code that is placed into the component where it can observe the internal state of components. There are three ways to place the sensors into components to track its state: manually insert the sensor code into the system during development, insert additional code by re-compiling the system together with the monitoring code, and detect the external system resource to predict the system’s behaviour.

The first one is manually inserting monitoring code by developers. When designing the components, developers insert the code around the sensitive interfaces to track the behaviour. This approach is usually used in the development phase, and it requires the developer’s efforts to design and implement the security features. For example, the log method is used to record behaviours during the run-time, so that the errors and failures can be analyzed and located from the logged information. This method depends on particular semantics of the system. The \( \text{log}() \) method needs to be inserted with many verbose \( \text{log}() \) statements at the location where it is required. When the requirements of the system change, the developers also have to be aware of the changes of tracking code as well as the locations where \( \text{log}() \) method was added.

The second method of placement is to insert additional features by the instrumentation of the source code. Byte code instrumentation is a method to weave separate track code into the existing program without modifications to the original program. The track code is considered as a separate security concern that cuts across multiple modules or the whole system. It contains the pattern expression indicating where to weave the code, what to weave and how to perform the actions. The weaving requires
recompilation of the system combining separate security concerns and the original system. The aspect-oriented programming is one approach to achieve this goal [42].

These two methods of adding monitoring features into components are performed during the system design phase perspective of the software’s life cycle. In most situations, the source code and internal structure of the components are not always available. How to obtain the run-time information of the components is a challenge as is to add additional features into existing components. The interfaces are the only positions we can collect the information among data exchanges between components. The basic pattern of such interactions can be modelled as an interceptor for capturing the messages to and from the components. There are also approaches of weaving aspects into existing software programs using bytecode instrumentation without access of the source code, like the aspect-oriented features provided by JAsCo [72], an aspect-oriented programming language tailored for component-based software system development. It provides the features of weaving aspects during class loading period.

The third method of monitoring is to use other third-party run-time monitoring tools [35] to assist with information collection. The information includes the resource usage, such as access control on files, performance behaviour, and memory consumption. This method of monitoring is to observe the external resources consumed by the target program. By analyzing the resource consumption of the program, the monitoring tools predict the potential harmful behaviours caused by the target program. However, it cannot reflect the real run-time behaviours of the program. The wrapping approach is more suitable to resolve the above two issues, by adding extra non-functionality features as extensions to the original component.

Similar to the first two methods, our work also uses the instrumentation to add
monitoring features into the component-based system. However, our approach adds a
wrapper to the component, and instruments the additional monitoring features into
the wrapper to monitor the associated component. A comparison of the first two
instrumentation methods and our work is shown in Table 2.1.

Table 2.1: Comparisons between Different Monitoring Mechanism

<table>
<thead>
<tr>
<th>Perspective</th>
<th>Manual Insertion</th>
<th>Instrumentation</th>
<th>Our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>source code</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>code separation</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>overhead</td>
<td>HIGH</td>
<td>LOW</td>
<td>LOW</td>
</tr>
<tr>
<td>applicability</td>
<td>development</td>
<td>development</td>
<td>released</td>
</tr>
<tr>
<td>events</td>
<td>any</td>
<td>operational events</td>
<td>method calls</td>
</tr>
</tbody>
</table>

2.1.2 Component-Based Software System

The component software stems from the popular programming paradigm Object-
Oriented Programming [73]. Component-Based Software System (CBSS) is increas-
ingly used in large-scale complex software systems. Components are encapsulated
into modules and used for the composition of a software system. It enables prefabric-
cated modules to be reused by assembling them into a new composite system. The
composite system is then called a component software system. The component can be
considered like a black box that we only know its provided interfaces and capabilities,
without the knowledge of the internal structure. The interface is the only way that
components communicate with each other. The component-based software system
is designed by assembling the separate components that satisfy the functionality re-
quirements, as illustrated in Figure 2.2. The users or system architects do not need
to know the components inner workings. What they do need to know is the interfaces
the component provides, what to pass in and what to expect back.
The well-known properties of components are the interfaces with its functionality specifications, including the signatures and the parameters. It can be a library, a modular program, or even a resource, such as a database system that provides public interfaces for user’s CRUD (Create, Read, Update, and Delete) operations. Replacing one component with another one that has equivalent interfaces and functionality allows the system to run with no need to rebuild it. It is considered as a unit of composition, including interfaces with specified contracts, and as well the context that the component depends on. The component should be able to be deployed independently and is also subject to composition by other parties.

![Component-Based Software System](image)

Figure 2.2: Component-Based Software System

Components are considered as the units of production that interact to form a functional system with specified contracts. Therefore, component-based development has two major aspects: components and composition. The component itself is a
regular software module, which could be composed of the other components (sub-components). Compared to other regular software programs, components play an important role in building large complex software systems. The component is an independent and reusable part of a system that accomplishes a system function, and works in the software architecture with a well-defined context. The system is working functionally by the interactions between the component interfaces. The composition is a method to connect all components to form the system. Appropriate connectors are necessary to assemble the components for the composition. Adding wrapper is an alternative method to change the way the component communicates with the rest of the system [33].

The system architecture or model acts as the context that allows the components to work and communicate within it. Component-based software systems are formed by selecting various components and assembling them together. The life cycle of component-based software development is independent from the life cycle of the composite system. The system requirements are carefully designed and evaluated with the perspective of the architecture. The system is divided into separate modules. The functional requirements for each component are identified based on the system requirements. The components developed by third parties or system vendors complying with the requirements are selected. The system architecture integrates the components into the system by using the appropriate adapters, connectors, and wrappers. The requirement includes identifying the specifications of the components and the definition of system architecture to permit the component’s collaboration [73]. After the components have been integrated into the system, the testing efforts are made to detect the failures with designed test cases from the component’s
functional requirement. Before the component is delivered to market, a component developer performs various tests, for example, unit testing to validate the functions and the basic security features of components. The integration testing is performed during the deployment phase to make sure that the components satisfy the functional requirements to form the global system.

The components are easily managed, replaced, and unplugged. This reduces the cost of software development compared to writing the modules from the scratch. The component models have been in development from different vendors, such as Java Bean [17], CORBA (Common Object Request Broker Architecture) [71] and COM (Common Object Model) [61]. There are also many open source projects that target toward the component programming model for flexible configuration and deployment such as: OSGi [19], Fractal [26], and Spring [79]. By building the system out of existing reusable components, it is helpful for the rapid development of large-scale software system in a systematic way.

The components are implemented as standalone with different security constraints in the context that is different from where it will be incorporated in. The reusable components may need an adapter to be properly integrated into the new environment. The components are interconnected by appropriate connectors [57] to communicate and exchange messages among them. The component is specified by its interfaces and the source code is usually not available when it is shipped, as many components are developed by third parties, independent of system development. A component is assumed to be an integrated part of a large scale application system, but the life cycle may not be compatible with the enclosing application system. Thus, the integrity of the components in the system is a potential risk that could possibly influence the
stability and robustness of the system.

The composition of the system needs to be consistent with global system in the security constraints and architecture design. The components containing bugs or flaws could be great risks to the system. Without the source code and the internal structures information, it is very difficult to ensure the quality of integrated components. Hence, monitoring approaches have become the desired way to assure the reliability of the component software system.

### 2.1.3 Component Wrappers

Wrapper is a technique of re-implementing the library, the class or even components, by translating the existing interfaces into compatible interfaces. New features can also be added into the wrapper, without the cost of changing the original components. The component wrapper is a specialized component for monitoring and filtering the data flows between the wrapped component. Wrapper is a general approach to ensure security and transparent component replications [58].

When components are selected to design a new system, the wrappers are developed for integrating components into system architecture. Using wrappers is a popular and cost-effective technique to integrate pre-existing components into the system. The communication between the selected component and the rest of the system can be altered in compliance with the system requirements. For example, the integer value returned from the component is changed to long double value with high precisions as required by system requirement. The translations are performed between two communicating components for argument formats. It is for the adapting
the functionality of components to system requirements. Wrapper has the advantage of refining the functionality of wrapped components without requiring detailed information about the internal structure.

In most cases, components are released and shipped without the access of their source code, especially in system designs with the COTS (Commercial Off-The-Shelf) components. The functionality and dependability of the components are usually insufficiently documented, thus causing potential risk to use the component relying solely on the component’s quality. The wrapper provides an option for system designers to compensate for the lack of the component’s information. By substituting the erroneous, suspicious returned values from components with new defences in the wrapper, the consequences of such failures can be mitigated.

The quality assurance of COTS system has been augmented by injecting middle layer software into the interfaces between the applications. The layer of the software then observes and modifies the data passing between interfaces. The additional security features such as encryption and access control are implemented in the layer of the software. The security functionality provided by the layer of the software adapts the selected components with sufficient security features to meet system requirement. In traditional cases, the application source code needs modification to implement additional security features to meet different system’s requirements, which increases the cost of developing various versions of components.

Wrappers were first proposed for use in hardware systems and has been used to adapt heterogeneous units. The same idea has been used in software to adapt existing libraries or modules into applications. For example, the DBMS (DataBase Management System) stored procedure calls are encapsulated into Java methods that
can be invoked by Java developers in EJB (Enterprise Java Beans) \[68\]. Therefore, Java developers can invoke Java method calls, without using SQL (Structural Query Language) statement calls to operate on the database. The wrapper approach eases the way that modules with different standards are integrated for the composition of the system. Wrapper can be considered as an re-implementation of acquired component, and also as an enhancement of the existing service.

The wrapper has been used to address various security issues by refining and filtering harmful messages \[67, 81, 56, 64, 25, 38, 58\]. The general purpose of wrappers is to insulate the external illegal requests from the wrapped instance \[58\]. The instance can be a functional library \[67\], a database procedure \[68\], or an instance implemented on a different platform \[56\]. In our work, we use wrappers to intercept the operational security events exchanged among software components. Wrappers separate the component computation and the way other components communicate.

2.1.4 Aspect-Oriented Programming

The traditional software development patterns integrate the functional concerns and non-functional features together during implementation. The logging mechanism is the typical example of such designs. The logging mechanism makes the debugging of system easier by locating the source of failures from the logged information where they occurred. However, such concerns are required in multiple places through the whole system, thus these are called cross-cutting concerns. Cross-cutting concerns are concerns that cut across multiple modules. We have been very familiar with the way of debugging by logging. The \texttt{log()} method is placed before and after the location you want to debug. There could be multiple places and modules requiring
trace and debug, then we also have to insert the `log()` method at those positions as well. The Aspect-Oriented Programming (AOP) technology provides the approach to modularize these cross-cutting concerns separately from the functional modules. Software developers can implement the non-functional concerns separately and weave them together with a functional code afterwards. AOP is an innovative programming model that enables the separation of cross-cutting concerns and it is convenient to use and to integrate into the development process. The basic concept behind AOP is that the hierarchical modularity mechanisms are inadequate to deal with concerns that need to be modularized but for which the implementation needs to be spread out over the application [70].

AspectJ [74, 42, 70], one of aspect-oriented programming languages, provides the constructs like point cuts, join points, advice, and aspects to help design cross-cutting concerns and integrate into software system easier.

Join points are locations where the extra code is weaved. Weaving is the process of integrating an extra code into the existing program at specified places. For example, in logging designs, the join point of logging concern is where the logging method is invoked, such as before, after or during its execution.

Point cuts are expressions of constructs that identify join points in programs. A point cut consists of join points expressions combined with logical operators. The logical operators are used to indicate the context of the join points, which are called point cut parameters.

Advice contains the set of actions to execute once the point cut matches the join points. The combination of join points, point cuts, and advice is called as an aspect. The aspect is encapsulated into an object in AspectJ. Advice defines additional
behaviours applied at the matched join points. When the execution of the program reaches the specified join point, the designated advice is invoked. In logging concerns, when the program reaches the method call, before or after, the logging methods are invoked automatically. Aspect is a well modularized cross-cutting concern which consists of point cuts and advice. The advice which is a reaction to join points, is associated with specified join points in an aspect.

An aspect is the essential element of AOP; it covers the non-functional issues with the position of the code and the actions taken at that position. Aspect weaving is a process of combining the aspect program and the primary program (functional program). The aspect code is compiled together with the functional code by the tool, aspect weaver, which is provided by the aspect-oriented programming language. The aspect code is inserted into the function code at the right place matched by the defined join points. After the weaving, a new system is formed containing both functionality and extra concerns integrated.

2.1.5 Design by Contracts

The Design by Contract (DbC) concept is initiated by Bertrand Meyer, the inventor of Eiffel programming language providing excellent support of assertions [60]. Assertions are executable checking code that is invoked during the execution of programs. The adoption of this paradigm presents an approach to develop robust and reliable object-oriented applications. A security contract is a type of mutual agreement for interacting objects, when neither of them has knowledge of internal details. The agreements specify the obligations required for client objects to interact with, including assumptions and expected results between objects. For example, Java Model
Language (JML) provides an approach of implementing contracts as annotations in Java 5. The **pre-condition** is the assumption requirement from the client to request the service, and it is specified with \@\texttt{requires} annotations; The **post-condition** is the guaranteed service from the server after finishing the request process, and it is specified with \@\texttt{ensures} annotations; the **invariant** is the status that should be maintained during the execution, protecting from the breakage of the service, and it is specified with \@\texttt{invariant} annotations. The annotations are interpreted and translated as executable code for run-time checks. In practice, the contracts have been specified as assertions in popular programming language such as Java and C++, Java Modelling Language (JML) \cite{53}, or embedded in the format of comments or annotations in source from Java 5 \cite{66,50,51}.

Execution Assertions (EA) is a specification language placed in specially-formatted and structured comments. The run-time checks are generated after the compilation of the program and embedded in the binary format of the system. The benefit of assertions following “design by contract” paradigm has been well known that it provides a flexible approach of run-time checks for testing a task. As presented by Bertrand Meyer in his design-by-contract method \cite{60}, the pre-condition checks are obligations on the callers and post-condition checks are obligations on implementers assuming that the pre-condition is satisfied at the time of invocation. The contract checking tools have also been ported in popular languages such as Java and C++. The \texttt{assert()} macros are provided in C and C++ as a primitive form of execution assertions \cite{66}. Java also provides many approaches to implement the “Design by Contract” methodology, such as the iContract \cite{50}, jContractor \cite{11}, Contract4J \cite{51}. Java Modeling Language (JML) also provides similar capabilities for Java language
JML is a behavioural interface specification language designed to specify contracts for Java classes. The assertions are expressed using annotations of \texttt{@requires} and \texttt{@ensures}. The compiler can then generate the corresponding checking code in-lined within methods. The JML also provides the other rich syntax to specify various assertions in comments, such as \texttt{@assignable} and \texttt{@model}. More details of assertions specified in JML can be found in [20].

2.2 Related Work

2.2.1 Security for Component Software

The major security concerns for component-based software is the inconsistencies between the components. The security requirements of an acquired component may not match with the system and thus the criteria varies in different degrees. The internal structure of the component is usually unknown to the rest of system. The interfaces are the only available information of the component for clients to collaborate with [63]. To eliminate the inconsistency issues, the necessary extra connectors and adapters are needed to integrate the component into the system [41].

The security consistency between components and the system cannot be predicted simply from only one single component. It is necessary to evaluate the environment context it is working in [54]. To better evaluate the quality of the software components, we not only need to measure the quality of each individual candidate components, but we also need to measure the quality of components when they are assembled to form the system and whether the composite system complies with the requirements [55].
During the stage of development, the individual components are evaluated and assessed using its unit testing and component’s quality requirement. To assess the component securities, black box testing was used to evaluate the every aspect that could cause the security issues on the component. The black box testing is a popular method to evaluate the quality level of components [47]. Black box testing is a functional testing by validating the behaviours of components from the external view. This method can only guarantee and monitor whether the implementation matches its advertised capabilities through interfaces. How to observe the securities of components is still being developed. In Khan and Jun’s work [48], a rating score system was proposed based on a template from security specification involving various security objectives, to assess the confidence level of components based on the black box testing results. The security objective profiles are built by evaluating the provided security functions, and assessing the captured security properties against the system-specific security criteria. Another testing approach of component security based on a dynamic monitoring and detecting algorithm during the development phase is presented in [31]. The approach is based on formal security specifications and the component’s capabilities. By evaluating the security concern testing results from dynamic monitoring, the extensions or enhancements can be applied to the components for later developments. The testing approach has also been used for monitoring the component’s behaviours [21].
CHAPTER 2. BACKGROUND AND RELATED WORK

2.2.2 Protective Wrappers for Security

Recent research in wrappers has addressed some important issues. Protective wrapper is a general approach to confine the failures of component, especially for COTS (Commercial Off-The-Shelf) components. It is a high-level approach to filter and protect the erroneous behaviours of components without changing their implementations.

Wrapping for dependability has also been addressed by other researchers. Wrappers are used to transform or filter the communications that may cause failures or errors. Fault injection has been used to identify such failure-causing values [43, 56]. Some component-based applications do not deal with kernel-raised exceptions properly. The wrappers can be used to protect them by transforming the exceptions into other formats as in a returned error code. It can also be used to protect components against inappropriate requests [43, 32, 38]. In [43], the library components are protected with wrappers to verify the failure-causing parameters passed in by malice. In [32], wrappers protect name servers from receiving unverifiable requests. Since the wrapper alters the interface behaviour of the component and the rest of the system, the designer needs to verify that these modified behaviours imply the required system behaviours. The COTS components are hardened by wrappers to form a solid system in [38].

Separating abstractions of connectors and computation of components increases modularity of component software system. Besides wrapping computation of components with extra functional features, connectors are also wrapped with trust management mechanisms implemented. The wrapped connectors check run-time behaviours based on trust level of incoming request records. Peter et al. proposes an approach of a trust management system in [46]. The connector of component is wrapped with
adapter to observe run-time behaviours and use Java security manager to control and report security events.

2.2.3 Aspect-Oriented Development for Security

The aspect-oriented programming technology was used to assess the well-being of the system. In [74], the concept of system health index was proposed as a measurement criteria of the internal system security state. The security concerns are encapsulated and added to the existing software with aspect-oriented technologies. In this way, the internal state of running processes and the exceptions thrown can be observed by the added sensor code (aspects) attached to the specific locations of the software. The approach uses the event-driven model to analyze the collected information from the sensor code. The aspects indicating the health of the system in this approach includes: consumption of system resources including processor cycles, memory usage, communication channels, and file handles; the number of thrown exceptions and the operational load of system, and the response time of a remote method invocation. The assessment requires the knowledge about the source code of the existing software system.

A security framework using aspects is illustrated in the work [84]. The security issues include the concerns of cross-cutting the component system, so that they could be addressed separately from the functionality concerns of the software. In this work, they treat the security as a cross-cutting concern and develop the aspects to enhance the security level of the system. By using the AOP to address the security issues in the software, it is more comprehensive and flexible to seamlessly integrate the security concerns into the existing functionality programs.
A secure software design with AOP is illustrated in [42], which presents how to encapsulate the security concerns and weave them into the design models using aspect-oriented programming. The aspects in terms of the roles are defined for different kinds of attacks. Three security objectives on access control are analyzed with different privilege to the data information, including confidentiality (READ), integrity (WRITE), availability (EXECUTE), to protect unauthorized information exposure. The aspects are designed for two role model types: Static Role Models (SRM) and Interaction Role Model (IRM). The static model defines the static structural properties of the model, and the interactive model defines the interaction patterns. The aspects of authentication are weaved into the design model to verify that if the identity claimed by the request client is legal and is an authorized identity. This approach shows a typical example to encapsulate the cross-cutting concerns into separate modules, and weave to the application to form a secure system. The only limitation of this approach is that it requires the availability of source code from the components.

The research presented in [65] uses the similar concepts such as wrappers, contracts, and aspects to ensure a component’s working in an untrusted environment. The aspect is instrumented into component system, dynamically adding wrappers around the method operations. The underlying component is changed to work with the external components. The approach adopts multiple wrappers to enforce contracts in different levels (outer and inner wrappers) that may obviously invite various overhead issues. The aspect is used to change the behaviours of the underlying components, which require access to the source of the components for re-compilation.

The aspects can be weaved into the existing system by the recompilation of source code. But when the source code is not available, the aspects weaving at bytecode
level is also possible as presented in [82]. The work demonstrates an approach of instrumenting additional concerns into a bytecode compiled Java program for mobile device. The aspects are written in pure Java programming language, and they are compiled together with the existing bytecode program without the access of the source code. The compiled aspects are weaved around the defined pointcuts using a reflection-based library, Javassist [9]. The library provides the expressive patterns of how to extract the join points from the bytecode. The aspect developer can only focus on the concerns without worrying about the complexity at the bytecode level.

2.2.4 Security Contracts

As the component-based software develops and becomes more popular, the contracts are also proposed to ensure the quality and security in components. The interface language IDL (Interface Definition Language) is introduced as the basic contract implementation and each component is required to implement the interfaces according to the specification [24]. The extra behaviour contracts are defined in the contract interface specification. Each designed component must implement the defined contracts so that it is strictly conforming to the interface specification.

The contract paradigm has been commonly used for run-time verifications in software [59]. The assertion is the early approach of embedding run-time verifications in the source via annotations [66]. The JML (Java Modelling Language) is a specification that can specify the contracts separately from the source code, while it also supports the embed into the code [53]. It provides rich features on defining various types of assertions as the contracts during run-time verification. The annotations are introduced into Java language since the Version 5 to support execution assertion
features [66, 50, 51].

The approach of annotations requires the efforts involved into the modifications on the source code or the compilation of the model and functions to generate new system containing the assertions. An approach of bytecode instrumentation of contracts is presented by Belle [23]. The approach of this work is similar to ours, but with some difference. The work by Belle focuses on enforcing policies around the execution flow, while ours focuses on the method call to verify that if the execution is safe to proceed, recording each actions. The work in [23] involves complex bytecode operation in the context of bytecode instrumentation using JVMPI (Java Virtual Machine Profiling Interface) and JVMDI (Java Virtual Machine Debugging Interface). This requires more strict semantic checking on instrumented bytecode. In our work, we implement the contract separately in pure Java language and there are no concerns about the semantic correctness and as well, the contracts are weaved with mature aspect-oriented programming language, such as AspectJ [2].

2.2.5 Security Monitoring

Monitoring the behaviours of the integrated components checks if the composition satisfies the system requirements, including functionality and security requirements. The software architecture can be decomposed into integrated parts such as the components, connectors, the system resources and the interactions between them. The constraints imposed on connectors can permit the secure communications between these parts. The interactions between the components are defined by the type of connectors [33], such as client-server protocols, pipes, shared memory, RMI (Remote Method Call), or RPC (Remote Procedure Call). The security mismatches could be
caused by the behaviour mismatches of the components. The non-secure interactions or the unauthorized access to system resources should be avoided in the system. Monitoring such mismatches can prevent the malicious exploits to the component-based software systems. Software architecture and components are closely related \cite{57,55}, and the monitoring on the system thus falls into how to monitor the individual components behaviours. There have been much research done working on monitoring components \cite{22,37,75}, but most of this research is primarily focused on verifying the functionality of the components, little research is concerned with security issues. The security monitoring works are summarized and categorized into the Table \ref{table}, including the objectives of monitoring, monitored information, monitoring mechanism, and the model of framework.

The complexity of the security model for each system specific context might be quite high, as presented in the work \cite{36}. It is based on the incremental design concept. To build a complete and perfect security model for the large-scale component software system, firstly the security system architectural model should be constructed; the system-level security constraints are then divided into the individual components; the verification of consistency between system-wide and individual components are required.

Built-in test approach \cite{21} refers to adding the verification contracts to the program so that the system can automatically perform run-time verifications. The built-in test uses the most straightforward method of assertion checking to assure the safety of the component. The verification code is written in the contract format, and is compiled into the binary code with the system. The assertion checking is employed in the development phase to assure that the developed components are providing the
requested services correctly in unit testing.
Table 2.2: Security Monitoring Related Work

<table>
<thead>
<tr>
<th>Work</th>
<th>Objective</th>
<th>Monitored Events</th>
<th>Methods</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>[21]</td>
<td>behaviours indication</td>
<td>security states</td>
<td>insert contract testing code</td>
<td>assertion checking</td>
</tr>
<tr>
<td>[22]</td>
<td>run-time verification</td>
<td>interaction message</td>
<td>proxy agent</td>
<td>specification comparison</td>
</tr>
<tr>
<td>[28, 29]</td>
<td>programming paradigm with monitoring</td>
<td>property violations</td>
<td>generate monitor code and recovery code in programs</td>
<td>formal specification and AOP</td>
</tr>
<tr>
<td>[31]</td>
<td>security testing</td>
<td>security states</td>
<td>monitoring on log data</td>
<td>detecting algorithms on logs</td>
</tr>
<tr>
<td>[32]</td>
<td>performance optimization</td>
<td>system resources related to performance</td>
<td>third-party monitoring tools</td>
<td>N/A</td>
</tr>
<tr>
<td>[33]</td>
<td>security model</td>
<td>architecture analysis</td>
<td>incremental modelling</td>
<td>imposing security constraints in incremental process</td>
</tr>
<tr>
<td>[40]</td>
<td>enhance traceability</td>
<td>trace message</td>
<td>code instrumentation</td>
<td>event-based model</td>
</tr>
<tr>
<td>[45]</td>
<td>security indication</td>
<td>error rates, time constraints</td>
<td>modelling by Petri nets</td>
<td>model comparison</td>
</tr>
<tr>
<td>[75]</td>
<td>service trustworthiness</td>
<td>credit records</td>
<td>trust monitoring algorithms</td>
<td>service-oriented</td>
</tr>
<tr>
<td>[80]</td>
<td>quality evaluation</td>
<td>quality related events</td>
<td>interceptor and low level monitoring</td>
<td>N/A</td>
</tr>
<tr>
<td>Our work</td>
<td>secure method call</td>
<td>interface properties and interactions events</td>
<td>wrapping components</td>
<td>aspect-oriented wrapper</td>
</tr>
</tbody>
</table>
Besides the testing approach, without too much knowledge of the internal details of the components, it is difficult to track the internal state of components, especially for COTS. Guo et. al. introduced a tracking framework into the component software to enhance the traceability of components [40]. A tracing code is inserted into components at the appropriate positions. Different types of trace events are defined for the components, and the events are registered to the listener of monitoring framework. Once the trace events are defined and the tracing code is inserted into the components, the monitoring framework is able to monitor the behaviours related to those events. The approach requires the code insertion during the component’s development. It is similar to the log mechanism to track the software’s status, and it also increases the code overhead since it requires the developer’s effort to manually insert the tracking code. If we can access the available source code of components, or if we can design the component-based system with common-contracts, the monitoring work during the run-time might be easy. But in reality, most components have no source code available for the monitor. In our thesis work, we encapsulate the component with a wrapper, and insert the tracking code into the wrappers instead of the underlying components. In this way, we could operate with the wrapper, isolated from the underlying components, keeping the components untouched. We capture the errors within the wrappers and confine them thus preventing propagation into the system.

Wang et. al. proposed an approach of on-line monitoring for Web Services [80].
The quality model is defined for multiple quality related events such as request/response message, management operations, resource state changes, and application execution. The application execution includes the internal component states and inter-component message events. To implement the monitoring framework, the agent code is instrumented into the target system so that it can provide the reflective interface which can be used to retrieve the important internal states information of the monitored components. The framework acts as an independent middle layer to collect the events during run-time, and the captured security related events are sent to a central analyzer for further decisions. By using a handler as the interceptor to collect the information such as request messages, response messages, method calls, and state changes on resources, the framework verifies the monitored events against the pre-specified constrains: value constraints for data consistency and sequence constraints for the business logic consistency. The analyzer also predicts the potential risks that could be caused by load overweights, memory leaks, and deadlocks, and then to react to system with appropriate adjustment.

Diaconescu et al. proposed a framework of automatically selecting optimized components during run-time environment [37]. The proposed strategy is to select the optimum components from multiple components with the same functionalities, based on the performance monitoring results. The run-time information in the execution environment is obtained, such as response times, throughput, incoming workload, CPU, and I/O usage. It uses a third party monitoring tool to provide run-time performance monitoring functionality. The collected information is analyzed and the potential problems are detected after the evaluation. The decision policies constitute the performance management logic.
Barnett et al. presented a way of inserting a proxy between two interacting components [22]. The proxy intercepts the requests and responses between the components. The proxy acts as a broker agent. It compares the observed behaviour with the specification model to detect the deviation of the components. It focuses only on the functionality consistency for run-time verifications. The request and response results can be verified by the proxy agent before delivering and returning to the components. Compared with this approach, our work moves the validation from the proxy between communicating components to the local wrapper of the component. In this work, a proxy agent is inserted between two communicating components, and the proxy performs the verification of the run-time behaviours based on the requirements, including type correctness, value correctness. But when the number of components grows, much more proxies are required to insert between those communicating components. For example, if there are five components interacting with each other, every two of them require one proxy in between in [22], thus it requires ten \( C_5^2 \) proxies, while in our work, the number of the wrappers is the same as the number of the interacting components which is only five. It only verifies the functionalities correctness based on the pre-defined model, but ignoring the security concerns verifications.

Grosclaude proposed an approach of a simple monitoring system based on the modelling of the external behaviour of software components using Petri nets [45]. Each component is associated with a local controller which observes the message received and sent by the component and compares them with the specified behaviour. During the execution, the monitoring system collects the error emission and the time constrains information coming from and to the components to reflect the state of the components. When the degradation of components, for example, a high rate of
error message emission, long component response time, or a complete blocking on a component is detected, then it indicates that a problem has happened on the component. The error messages are located to analyse if the component contains the critical-execution program causing the slowdown of the process. It is a typical monitoring system based on a behaviour model by defining the normal messages exchanged between components. The abnormal messages emitted between components can be located to predict the source of problems.

The trustworthy component framework was proposed to make the component contract aware in [75]. In this work, suspicious components would be prevented from any malicious requests. The framework is involved during the software development phase, and it is a learning process that requires the history records to track the trust level of the requesting components. From the individual components point of view, the problems still occur when the trustworthy component happens to be injected by malicious code. The trustworthy component is then manipulated by the attacker, due to the high credit of its trust level it already achieved. Therefore it also passes the authentication validation to request the service, which is a potential risk.

Monitoring the component from the other aspects has also been researched to indicate the state of the component system, such as performance monitoring [37], run-time function verification [22]. The performance-monitoring focuses on the system resource monitoring. The system resource changes reflect the state of the components consuming the resources. The run-time verification is primarily to check the interface functionality of the components, but it does not address the security issues, such as detecting unauthorized requests or compromised components. Without the detailed information of internal structures of the component, it is difficult to locate where the
flaw statement is. For example, one secure function of the component could be unsafe in another system. Therefore, to better describe the safety of the components, we need to consider more evaluations on events together with the context of the enclosing application system.

To improve the quality of the developed software programs, a formal framework, Monitoring-Oriented Programming (MOP), was proposed by Grigore Roșu [29, 28]. The framework uses the paradigm combining the specifications and the implementations. It is similar to developer’s embedded assertions and runtime checks, but with more complex formal logics and with automatically generated monitors for specified properties changes. The corresponding development tools have been implemented to support the MOP paradigm [27, 30]. During development, the developers declare the properties using high-level formalisms, and the corresponding monitor is generated to detect the violations on the property based on the declared formal specification. It is a good way for developers to include the monitors to assure high quality of the system. It is applicable to in-house developed software programs, but not to acquired software components, such as COTS.

2.3 Summary

In this chapter, we have introduced the background knowledge of component-based software system along with its security issues, wrappers for components, the aspect-oriented programming technology, the security contracts, and the monitoring system overview. The related work in the above area have been presented in the following categories: security concerns on component software, wrappers to protect error-prone components without touching the source code, the aspect-oriented paradigm to weave
additional security features to compliment the component software, the security con-
tracts to protect the system at run-time and security monitoring on component soft-
ware. Taking benefits for each of areas, we propose the aspect-oriented protective
wrappers with contracts to monitor the behaviours of the wrapped components.
Chapter 3

Wrappers for Component Monitoring

In this chapter, we illustrate component wrappers and the approach of designing the wrappers. We associate a wrapper to each underlying component to intercept and observe transactions between the wrapped components. Security events between wrapped components are captured by the wrapper using both adapter and proxy patterns. The wrapper delegates harmless requests, filters suspicious input data and ensures that the incoming request is processed without harmful effects on components.

Section 3.1 presents the overview of the monitoring framework. Section 3.2 presents the overview of using wrappers and aspect-oriented security contracts. Section 3.3 illustrates the design and implementation of component wrappers. Section 3.4 shows an example of using component wrappers to monitor the invalid inputs. Section 3.5 describes the monitoring process of the wrapper approach. Section 3.6 summarizes this chapter.
3.1 Monitoring Overview

Monitoring has been in popular use to observe the application’s run-time behaviours, such as tracing, logging, and performance analysis. Some monitoring systems analyse run-time behaviours by observing the usage of external resources such as file access, memory consumption, and I/O throughput time [37]. Modelling has been introduced in the early phase of software development life cycle to add monitoring features into software systems [45]. It is also a popular method of embedding testing features so that the system has built-in test functions [21]. Instrumentation is a technique used to embed monitoring features into compiled and deployed software systems [76]. These monitoring approaches observe the system’s behaviours either by external resource consumption changes or by heavy modifications against the original system. However, the Component-Based Software System (CBSS) consists of acquired reusable modules, which are only known for their provided service interfaces, without exposing other internal information. The modules are also shipped without the available source code. These unique characteristics of CBSS make these traditional approaches not applicable, as it has neither access to source code to make modifications, nor the internal information about how it consumes resource usage. Therefore, the monitoring of transactions between components becomes a desirable way for run-time quality assurance on components.

In our work, we propose a monitoring framework based on wrappers. We design the wrapper for each observed component to intercept the communication between the component and the rest of the system. The implemented wrapper is deployed together with the underlying components. The evaluations of the run-time events are performed by the proposed monitor. The monitor consists of aspects and contracts.
The aspects define the rules specifying where the monitor is placed and the events that the monitor should capture. The contracts implement the mutual obligations as verification functions that should be assessed against the context specified by the aspects. The wrapper-based monitoring framework presents a realistic view of observing the run-time behaviours of interactions among components without modifications to the underlying component. The anatomy of the monitoring approach with wrappers, aspects, and contracts is shown in Figure 3.1.

Figure 3.1: Anatomy of the Monitoring Approach

**Component Container**

The *Component Container* is the environment in which components are deployed, for example, the Java Virtual Machine (JVM) running Java compiled classes (component in class-level), web browser for Java applets, web container running JSP (Java Server Page) and Servlets [8], the BeanBox toolkit for JavaBeans, and application server for Enterprise JavaBeans (EJB) [17] such as EasyBean included in JOnAS application server [12] and EJB container in JBoss application server [10]. The *Component
Container manages the life cycle of the deployed components and is responsible for connecting assembled components to form a composite system runnable in the container.

Security Policy in OCL

To enforce additional security constraints into the system, the policies are declared in OCL (Object Constraints Language). The policies defined in the OCL format includes the security contract to evaluate during the execution, and the monitoring aspect to instrument the contracts into the component wrapper. The policies in OCL include the context information where the constraints are enforced, the implementation of security contracts that accomplishes the task of run-time verification against the context, and the aspect specifying the join points to weave the contracts into the context.

Aspects & Security Contracts

There are two kinds of aspects used to develop the monitor system. The Aspects for Wrapper define the aspects to deploy the wrapper into the system. The deployed wrapper is then associated with the monitored component. The Monitoring Aspects define the aspects to instrument the security contract into the wrappers. The Security Contracts define the functions of evaluating the intercepted method calls. It is a functional module that is inserted into the wrappers via aspects. The contracts evaluate the safety of the monitored method call. The wrapper intercepts these method call events and passes the contract module the context information such as method name, origin of method calls, and the input data. The contract module
verifies if the context information comply with required system specification. The monitoring aspect can use different contracts to evaluate certain kinds of events, such as method calls. The details of weaving security contracts with aspects are illustrated in Chapter 4.

The additional features are added around the method invocation on the underlying components from within the wrappers. These features are the extra security concerns that we intend to integrate around the monitored events. The additional concerns can be added manually in the wrapper designs, and can also be instrumented into wrappers around the invocation using aspect-oriented technology. In our work, we use the latter, which is to implement the additional concerns as a separate module and integrate into wrappers as a cross-cutting concerned aspect. The advantage of this approach is that we can specify the security policies separately, and the security concerns are implemented based on security policies and can be plugged flexibly.

Component

The Component is encapsulated into a restricted context managed by the wrapper. The wrapper performs the run-time checks before the method is executed to assume that the incoming request is valid, safe and sanitized. The wrapper detects violations of returned value of the method invocation to guarantee that the post-conditions of the method call are compliant to system specification. The internal state of the component is also checked during the execution of method calls. All incoming requests to underlying components are intercepted by wrappers. The wrapper forwards or filters the requests based on security check results performed by the security monitoring code, such as parameter check or roles check. Therefore, the components are
protected by the wrappers with the instrumented monitoring code. The wrapper communicates with the rest of the system on behalf of the wrapped component as a proxy agent.

**Wrapper**

The *Wrapper* encapsulates the monitored component as an extra protection layer without modifications to the underlying component. With the wrapper, the component’s behaviour can be monitored via its wrapper interception and delegation. The wrapper performs two major functions for the wrapped component: first, it wraps the component without changing the functionality of the component, although it can enhance the functionality by adding extra features; second, it acts as the proxy to delegate any incoming requests to the component. An adapter pattern is used to generate the wrapper, which inherits all the functions of the underlying components and also includes the reference to the components. The reference to the components is therefore the association relationship between the wrapper and the wrapped component. The wrapper can retrieve the status of the component by inspecting the associated component via the introspection on the component. In our wrapper design approach, the proxy pattern is also adopted. Proxy mechanism is a way to automatically wrap the underlying components and allow the wrapper to intercept and delegate method calls to the component. Combining the adapter and the proxy pattern, the wrapper has access to component interfaces and can confine the events between the component and the rest of the system. The wrapper is constructed around the target point where the component was constructed and is deployed on behalf of the components. After the wrapper is deployed, any subsequent method calls to the wrapper component will
be dispatched to the wrapper. The wrapper executes the run-time verification implemented in security contracts, and the violated security events are observed, logged, and stopped from execution.

### 3.2 Enforce Additional Policies into Wrappers with Aspects

The wrapper is the fundamental layer on which the monitoring framework is built. After the wrapper is constructed, the additional monitoring features implemented with security contracts are instrumented into wrappers using aspect oriented technology. The aspects are specified in a security policy language together with the reference to the security contract implementation. For example, the aspect of `SQLInjectionAspect` contains the reference to the contract of `SQLInjectionContract`, so that the contracts are executed around the target point specified by the aspect.

The adapter and proxy patterns are used to design the wrapper. Therefore, all requests on component interfaces are intercepted in wrappers because of the proxy feature. The simplest wrapper as a proxy agent, without adding features, will simply forward the invocation to the underlying component for execution. However, as the wrapper is designed in an adapter pattern, it is also possible and feasible to add extra functions to the wrapper, which is the contracts weaved by the aspect.

Constraints are specified in a security policy language. In our work, we use the OCL (Object Constraints Language) to specify the contracts, as it describes the enforced obligations between collaborating objects. The aspect is extracted from the constraints, and it specifies the target point where the monitoring should be
performed.

The security contracts are implemented as a separate module independently from the system and the wrappers. For example, a SQLInjectionContract is implemented to verify if the received parameters contain the illegal SQL statements or keywords. This is independent of the context of system. It can be used in multiple locations as specified by the designed aspect.

The security contracts are instrumented into the wrappers via specified aspects. Multiple contracts and aspects can be instrumented into wrappers to monitor different events. The rules are evaluated upon method calls of the underlying components captured by wrappers. From the perspective of a monitoring system, the method call is the subject under observation. The sensor of the subject is the association relationships of component and its wrapper. The wrapper has the access to the wrapped component via introspection and inheritance from the component. The wrapper is able to capture all events to the components. The monitoring framework can select and filter events according to the user-defined expressive patterns in aspects. In the aspects, we define what events to intercept: method call, or a thrown exception; when to capture it: before or after method calls; and how to respond to the events: invoking the contracts referenced in the aspect. According to the instrumented rules of security contracts, the wrappers evaluate the event information and forward or deny the request accordingly.

In this chapter, we focus on the wrapper design and the wrapper implementation that is used for monitoring. We will discuss the enforced security concerns in contracts and aspects in Chapter 4.
3.3 Design and Deployment

3.3.1 Overview

The purpose of wrapping the component into a black box is to control the behaviours under the system specifications. The original integrated component may have flaws or design errors, which cannot easily be managed and controlled. Confining the external behaviours of the components is an assurance of a safe environment for enclosing system and verifying the quality of components. The wrapper is an enhanced re-implementation of the existing components. The access to the interfaces and properties of the underlying components are available from the wrapper. The wrapper’s interfaces are defined based on system architecture, so the interfaces and properties of wrappers are equal to or less than the actual selected components. Otherwise, the selected component does not satisfy the functional requirement, since it lacks required interfaces and mismatches with system requirements.

By wrapping the component, transactions among components are under observations of the wrapper. The wrapper inherits all functionalities from the component, and contains a reference to the underlying component instance. Thus it acts as the delegate of the component to forward the calls to the rest of the system, and as a filter to sanitize the suspicious requests to the component. The component interfaces are the same as the wrapper’s interfaces as is illustrated in Figure 3.2. The wrapper not only has access to all the information from the components, but also can customize the functionality inherited from components, including blocking some interfaces and adding more interfaces in wrappers. As the wrapper is designed as an extension to the original component, the fields of the underlying component can be also accessed.
from the wrapper.

The often referenced definition of component is provided in [73], that “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only”. Based on semantics, the current software component models can largely be divided into two categories: models where the components are classes, and the other that models where the components are architectural units, as in software architectures [52]. We use the first category of component models in our work, the modularized classes. The example of component models where components are classes include JavaBeans, Web Servlet and EJB, since they are special Java classes hosted by component container, Bean Box and EJB container respectively.
3.3.2 Design and Implementation

Wrapper is basically another component that has access to all interfaces and properties of the wrapped component. We use the adapter pattern and proxy pattern to design the wrapper for the purpose of monitoring.

A class modelled component is the component module implemented in a fine-grained class. It can be a single Java compiled class, a JavaBean for GUI (Graphical User Interface) applications, a servlet on web container, or even an EJB (Enterprise JavaBean) class that is deployed and managed by application servers. A class modelled component is the fine-grained component in format of the compiled class that has no dependency on the other classes.

We use an adapter pattern to wrap the component’s functionality. The wrapper is another component dependent on the wrapped component. As the adaptor to the underlying component, it contains the reference to the component and it is able to invoke the methods to the original component.

The intention of the adapter pattern is to “convert the interface of a class into another interface that the clients expect and it lets classes work together that could not otherwise because of incompatible interfaces” [39]. We adopt this pattern in the component wrapper to make sure that the interface of the underlying components can be consistent with the system specification. If the acquired component lacks an interface, the required interface can be added into the component wrapper ensuring the adaptability.

We also use a proxy pattern to delegate the request to the wrapped component. The proxy mechanism is used to invoke all method calls to the wrapped class from the wrapper. The proxy features in Java language are used to retrieve the information
from the class via reflections and run-time introspection. By introspection on the class, the interfaces and property information can be accessed from the wrapper.

Combining the two patterns we generate the wrapper that contains the reference to the underlying component and can intercept and forward the incoming request to the wrapped component. The Figure 3.3 illustrates the overview of the class modelled wrapper. After wrapping the component, all method calls and its public interfaces are intercepted and delegated by the wrapper for invocation. The selection of the captured method calls are performed by the interceptor, filters, and the defined constraints.

![Figure 3.3: Process of Method Call in the Component Wrapper](image)

By default, the interceptor captures all method calls in the proxy mechanism. The filter is designed in the format of aspects, and it specifies the expression pattern of selected method calls, and the constraints are the contracts that are weaved into the wrappers by the aspect. After the constraints are available to execute, the wrapper becomes the component with self-built-in run-time verification features.
3.3.3 Deployment of Component Wrapper

As the wrapper is acting as the proxy agent serving the system on behalf of the wrapped component, the re-deployment of wrappers with reference to the wrapped components is needed to form the monitoring system. However, the deployment only happens when the original component is constructed. Instead of returning the reference to original component to rest of the system, the reference to the wrappers is returned to the client who requested that component service.

The wrapper is generated using an adapter pattern, thus the returning type is the same as the original type. The only change is the returning reference value. We use the aspect-oriented technology to accomplish this re-deployment. The aspect specifies the target point where the original component is initialized. The advice defines that before exiting the constructor of the original component, the reference of constructed component is passed to its wrapper constructor to return the instance of the component wrapper.

As is illustrated in the Figure 3.4, when the client invokes the component’s service, it first calls the component’s constructor to get its reference. The component is then executing its `new()` method to return the reference of the component instance. Around the construction, the wrapper is generated with the component instance passed in as the parameter. Instead of returning the type of component, the wrapper is returned during the component’s construction. After the wrapper reference is returned, as the wrapper inherits all functions of the underlying components, any subsequent method calls against the component are still available to execute with the returned wrapper reference. In this situation, wrapper is considered as the subtype of the underlying component, so only instrumentation around the constructor is necessary to accomplish
the re-deployment, thus reducing the system changes to the minimum.

![Sequence Diagram of Wrapper Deployment](image)

Figure 3.4: Sequence Diagram of Wrapper Deployment

The aspect illustrated in the above flow diagram is in detail in Figure 3.5. We use the around advice to return the new value of the wrapper, instead of the component. Firstly, we specify the pointcut of component’s construction. At this target point, the arguments and the instance variable are available to access in the context. Then the around advice is used to return a new type of component wrapping the original one. The <<Component_Class>> is the place holder for component’s class type. The <<Client_Class>> is a place holder for the client component from which the component’s service is requested. For example, if we want to monitor the component
usage from a specific module, then the re-deployment is only needed in this module without touching the others. The <<Wrapper>> is the place holder for wrapper’s type. The returning type of <<Component_Class>> is specified for the around advice. As the wrapper is designed with the adapter pattern, so the <<Wrapper>> type can be returned with all functions inherited from the component. The bind method is used to register the association relationship between the component and its wrapper.

```java
public aspect DeploymentAspect{
    pointcut ComponentConstructor(<Component_Class> aComp);
    call (<Component_Class> .new(args)) && this(aComp) && within(<Client_Class>)

    <<Component_Class>> around ComponentConstructor(<Component_Class> aComp):
    return new <<Wrapper>().bind(component);
}
```

Figure 3.5: Deployment Aspect for Component Wrappers

### 3.4 An Example Application

We will use an example application to illustrate the wrapper design for monitoring. The example application consists of a model component, which is a pure functionality module, and a client GUI (Graphic User Interface) component, which calls services provided by the model component.

The model component is implemented as a service module that can be reused in different places without modifications. For example, as is illustrated in Figure 3.6, a component named TemperatureConverter is designed to calculate the temperature in Celsius and Fahrenheit units. The component is an independent module incorporated in the system.
The component provides the computation and conversion service to the rest of the system. The component takes the functionality as its main concern without the other constraints, for example, the range of temperature allowed to be entered. Different clients using the same component may have different requirements, as is illustrated in Figure 3.6. Two clients are using the TemperatureConverter service, the GraphGUI and FahrenheitGUI client components. In GraphGUI component, it adopts the TemperatureConverter to display the temperature of water with the range of 0°C to 100°C, while the FahrenheitGUI client only uses the TemperatureConverter component to form a completed application as the front end GUI and back end computation service. There are no constraints required on the range of temperature values entered in the text box field. The requirements of GraphGUI client component allows only values within the normal range be displayed reflecting the current water temperature. Thus, additional constraints should be added against the adopted component model,
otherwise, abnormal results could be explored. As shown in Figure 3.7, the figure on left hand side is the normal behaviours of the GUI application within the allowed range of temperatures. On the right hand side, as the application does not intend to deal with temperature values beyond the allowed range, the abnormal behaviour occurs when the input value falls in illegal ranges beyond the expected valid range.

![Diagram](image)

(a) In Normal Range  
(b) Not in Allowed Range

Figure 3.7: Incorporating TemperatureConverter Component without Wrappers

### 3.4.1 Implement the Wrapper

As the TemperatureConverter component is only used for the conversion computation, no security or other concerns are considered for the context in which it will be used. Without modifying the component, the wrapper is implemented inheriting all the functionalities from the underlying components. The additional concerns are then added into wrappers as enhancements.

To implement the wrapper for the monitored component, we use adapter pattern
and proxy pattern together, to inherit all features from the underlying component and to intercept all requests to the destined component. The UML diagram of the component wrapper is shown in Figure 3.8.

By default, the wrapper inherits all the interfaces from the TemperatureConverter component using an adapter pattern. The additional interfaces can be added into the wrapper by defining new methods for the wrapper. The InvocationHandler is an interface that a proxy agent must implement to intercept the requests to its delegated component. Therefore, as shown in the diagram, whenever the method call is invoked on the underlying component, which is the instance attribute in the wrapper component, the request is firstly intercepted by the wrapper, and then the method invoke is called to execute the method to proceed the execution.
3.4.2 Implement Monitoring Code

Policy is the constraint we need to enforce upon the events that we capture. For example, we have a feature request of monitoring each method call to the underlying component, then we can specify a logging policy to be enforced around each call to the component, in our case, which is around the `invoke` method.

To add simple logging monitoring policies, we enforce it around each method call to the underlying component. As mentioned above, each method call is intercepted by the wrapper and followed by the `invoke` method call, thus the logging action can be added around the method of `invoke` in the wrapper, as shown in Figure 3.9.

```java
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    // TODO Auto-generated method stub
    // log the setter and getter methods
    Logger.info(Level.INFO, "calling " + method.getName() + " methods ...");
    method.invoke(instance, args);
    // log the setter and getter methods
    Logger.info(Level.INFO, "exiting " + method.getName() + " methods ...");
    return null;
}
```

Figure 3.9: Logging Policy for Component Wrapper

Line 1 and line 2 are implementing the method `invoke` that is required by the interface implementing the proxy pattern. The parameter `proxy` is the delegate that processes the methods on the wrapped component. The `method` is the intercepted method call object. The `args` stores all the arguments of the method call. Line 7 executes the intercepted method call on the underlying component. The `method` is the intercepted method call object. The parameter of `instance` is the instance object of the underlying component. It is one property saved as a reference in the wrapper.
Line 6 and line 9 is the logging policies enforced before and after the execution of the method call.

We can also enforce the value constraints around the intercepted method call. For example, for the setter methods, an additional verification is required on the parameters before proceeding the execution. If we want to verify the parameter value of the method setF() is in the range of 32 to 100, we can have the verifications added for the invoke() method in the wrapper.

```java
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    // TODO Auto-generated method stub
    if (method.getName() == "getF") {
        double parameter = (double)args;
        if (parameter >= 32 && parameter <= 100) {
            Logger.info(Level.INFO, "The parameters are verified: \[" + parameter + "]");
            Logger.info(Level.INFO, "calling " + method.getName() + " methods ...");
            method.invoke(instance, args);
            Logger.info(Level.INFO, "exiting " + method.getName() + " methods ...");
        } else {
            Logger.warning(Level.INFO, "The parameters are not valid: \[" + parameter + "]");
            Logger.info(Level.DEBUG, "The expected range is between [32,100]");
        }
    }
    return null;
}
```

**Figure 3.10: Enforce Validation Policy for Component Wrapper**

Figure 3.10 represents the implementation of the enforced policy. Line 5 is performing the check on the intercepted method call by verifying its signature name. Line 7 to line 11 record the event information in logs if the parameter passed in is in the valid range and proceeds the execution. Line 13 and line 14 record the event information in the warning message with the captured data.
The parameters can be retrieved in the context of `invoke` method via the `Method` class using reflection features. If the parameter falls into the allowed range, the execution is proceeded, otherwise, the warning message is recorded that the parameter is not valid, the execution is skipped and returned.

We can see now that the additional monitoring code is inserted into the wrapper manually. This is not a good solution for enforcing multiple security policies. In our work, the security policies are further defined in OCL (Object Constraints Language) which can be weaved into the wrapper automatically, and multiple security policies are enforced by aspects: more details regarding this will be given in Chapter 4.

### 3.4.3 Deploy the Wrapper

After the wrapper and policies are inserted, the wrapper needs to be deployed into the system. In this example, we need to deploy the wrapper reference to the clients that use the `TemperatureConverter` component service. We know that there are two clients using the component service, so the method call comes from within these two clients. We deploy the reference at the places where the component constructor is called from these clients. In these target points, we pass the constructed component reference to the wrapper’s constructor, and then return the wrapper’s reference instead to the client. The subsequent method calls to the components are not changed, because all features of the component are also available to access via the wrapper as well, thus the integrity remains.
Figure 3.11: The Aspect for Instrumenting into the new Wrapper

Line 2 to line 4 specify the pattern of the intercepted construction of the component. It means the locations where the component is constructed and the constructor is called from the GraphGUI or the FarenheitGUI component. Line 6 and line 7 specify an around advice when the point cut is satisfied. It passes the constructed component to the bind() method in its wrapper component. The method returns a proxy object as the delegate to its wrapped component. The wrapped component is saved as a reference in the returned component wrapper.

The patterns specified in aspect are shown in Figure 3.11. The returned wrapper component is an encapsulation of the original component. It inherits all features from the underlying component, and also implements the proxy features, so that it can act as default proxy agent to request service to the underlying component.

3.5 Monitoring Process of Wrapper

Wrappers are designed as the outer layer of a component, in which additional behaviours can be added as enhancements. In this section, we describe what behaviours are available for wrappers to assist monitoring, and the mechanism of how the wrapper processes the events for monitoring. Section 3.5.1 presents the run-time verification
behaviours that wrappers can incorporate into monitoring, and Section 3.5.2 describes the mechanism and the flow process of how the wrapper performs the verifications.

### 3.5.1 Run-time Verification of Behaviours in Wrappers

The traditional method to monitor the components is to build a reference model, and compare the run-time behaviours observed with the expected behaviours as specified in the model. However, building a perfectly secure model is difficult, because the enclosing application environment can be quite different from where the individual parts were developed, and the composition of security architecture could be quite complex depending on deployment environments.

In our work, we target on monitoring component interactions to automatically verify component’s run-time behaviours. Because a component is used only by its exposed public interfaces, and while wrappers can selectively re-publish its interfaces, the functionalities can also be enhanced by wrappers via implementing extra features in wrappers, such as logging, tracing, data sanitation and access control, complementing the potential flaws brought by underlying components.

The unique feature of the component-based software system is the easy assembly of the system by integrating pre-fabricated components. In our research, the monitored components are integrated into the system by its wrapper. Therefore, wrappers are deployed into the system communicating with the entire system, instead of only communicating with the underlying components. The system is assembled by wrappers with each corresponding component implementations associated. The business logic, operations, and data access are performed through the wrappers with the wrapped components hidden inside.
Figure 3.12: Wrapper Monitoring Process
The model of wrappers is extracted from system’s functional requirements. Wrappers separate the role of connecting with the rest of the system and the role of computation of component instances. It acts as the role of mediating communications and interactions among components and coordinating the transactions for the services provided by components. In this phase, wrappers manage how to provide the public interfaces and how to forward the requests to services of wrapped components. The computations of services are implemented in wrapped components, and the interaction is implemented in wrappers.

Moreover, wrappers evaluate the status of each incoming request and returning results to detect a malicious service request, such as incompatible parameters containing sensitive script or SQL statement data, and exploit hidden information data. Wrappers specify auxiliary non-functional mechanism to govern the communications between wrapped components. The non-functional mechanisms include security features like logging and access control. The security monitoring rules are implemented as separate modules and instrumented into the wrapper’s model to form a composite component-based system with built-in monitoring features in component wrappers.

The state machine diagram of the process of wrapper monitoring is shown in Figure 3.12. The wrapper is deployed into the system on behalf of the wrapped component. When the system invokes the method call provided by the underlying components, the request is intercepted by its wrapper. The wrapper evaluates the request context according to specified monitoring rules, for example, format of data exchanged, and the correct type of parameters. The wrapper starts to perform the evaluations against the captured method calls, the exception thrown from the component, and the access to the component’s properties. It first assures that the incoming request meets the
requirement of pre-condition to proceed with the service. If it passes, the method call requests are forwarded to the underlying components for further execution. Otherwise, it refuses the request and notifies the event information. During the execution, the wrapper will observe the state changes of a component, such as the invariants state in component. If any changes violate the security rule, the wrapper will abort the execution and notify the event information. After the actual computation by the component is finished, the wrapper will assess the result to ensure its safety to return. The completed flow of wrapper verification is an assurance to the secure method call so that components are not compromised by any possible malicious actions.

In the example described in section 3.4, after wrapping the TemperatureConverter component, the request to the underlying components are intercepted by its wrapper. The wrapper invokes the additional constrains defined in a separate module to evaluate the context, for example, checking the range of received values of temperature. After the constraints are enforced into the wrapper, the built-in monitoring features serve as the run-time verification during the execution. An example of such verifications using contracts will be illustrated in Chapter 4.

### 3.5.2 Event-based Monitoring by Wrappers

A completed monitoring framework consists of an entry, a sensor, and a response mechanism, as was introduced in the monitoring tutorial work [69]. The entry is the subject under monitoring, which is the wrapped component. The wrapper encapsulating the component is registered as the listener on the component. The listener subscribes to the state changes on the target entry, such as method calls invoked, property fields modified. The changes are notified of the listener, which is the component wrapper.
The events are then evaluated by the monitoring wrapper against the method call and the context information. The corresponding actions, which are implemented in the aspect advice, are executed in the context as the response to the events. The process flow of wrapping components, intercepting method calls, invoking defined aspects, and returning the evaluated results is shown in Figure 3.13.

Figure 3.13: Process Flow of Evaluating Intercepted Method Calls

Introspection is used by wrappers with reflection capabilities of the component to automatically retrieve the interfaces and property information. Those interfaces to be monitored are registered in the wrapper. To make the components observable with more flexible rules, we deploy the components with the wrappers, so that the
component can be plugged into the wrapper or unplugged from the wrapper. The component is hidden inside the wrapper when deployed into the system. When the event of invoking method call to the component occurs, as the method call has been registered with the wrapper via aspect, the method call is delegated to the wrapper for process. The wrapper receives the method call information such as the signature of the method, the parameters, and the origin source of the method call, and so repeats the process. The monitoring rules, implemented in format of security contracts, are evaluated against this information from within the wrapper. When the verification proves that it is safe to process the request, the wrapper proceeds with the method call and forwards it to the underlying component. Then the component is responsible for the actual computation only and return the computed results.

3.6 Summary

We demonstrated the monitoring approach with wrappers. In our approach, we designed wrappers to manage and govern transactions between the wrapped components. The wrappers are implemented using an adapter pattern and proxy pattern. Wrappers manage properties and interfaces of the wrapped components. The wrapper can be enhanced with the added security monitoring code. It separates the task of confining the component’s behaviours and enforcing security policies specified separately, without modifying the implementations of the wrapped component. This approach increases the re-usability of the monitoring framework to enforce same security policies on different systems with the same functionality requirements, but with different acquired components. In the next chapter, we demonstrate the approach of weaving security contracts into wrappers with aspects.
Chapter 4

Security Contracts in Aspects

In this chapter, we will cover designing security contracts for component wrappers and generate the aspect code to instrument into the wrapper. Section 4.1 briefly introduces the overview of security contracts used for run-time verifications. Section 4.2 illustrates the approach of instrumenting the security contracts into the component wrapper as the aspect code. Section 4.3 shows using aspect oriented methodology to weave the security contract as separate aspect code into the component wrapper. The integration of the monitoring framework with the aspect-oriented wrapper is demonstrated in Section 4.4. Section 4.5 concludes this chapter.

4.1 Security by Contracts and Aspects

With the designed wrappers, additional functions can be inserted around the execution of requested method calls on the underlying component. Instead of inserting the additional concerns into wrappers manually, we propose the approach to weave the security contract into the wrapper via aspects.
Security contract is an implementation of the security constraint imposed on the interaction of two communicating objects. In our example the constraints are mutual agreements enforced on the interactions between the involved components. In our example, the constraints define the pre-condition, post-condition, and invariant checks around the method call. It ensures that the method call satisfies the required condition to be forwarded to underlying components, such as that the properties of component are not changed during execution, or the returned results are not modified before received by the client.

The aspect is defined to specify the pattern of locations where the security contract is weaved into the wrapper. The join point defined in AspectJ has the counterparts to the conditions defined in the contract, like the before advice indicating the pre-condition contract. The contract, as the implementation of the security policy, is thus enforced in the wrapper via the aspect-oriented instrumentation.

The sequence diagram of monitoring components with weaved security contracts is shown in Figure 4.1.

The contract module ConstraintsContract is the implementation of security policy to impose on the incoming method call to the Target component. The Target component here represents the wrapper together with the wrapped component. The requested method call is firstly intercepted by the wrapper and then is proceeded via the invoke() in the wrapper. As the contract is weaved into the wrapper by aspect, the instrumented code in the wrapper contain the reference to the contract implementation. Thus when the wrapper receives the requested method call specified by the aspect, the defined contract is called to evaluate the current context of the request. When the Client component sends the request to the Target component,
the ConstraintsContract is invoked with the context information passed in.

![Sequence Diagram of Using Contracts for Component Monitoring](image)

Figure 4.1: Sequence Diagram of Using Contracts for Component Monitoring

The ConstraintsContract performs pre-conditions checking and proceeds to execution if the pre-condition is assured. The invariant is conducted during the execution of the method call, if the violations are found during the invariant checking, the returning value is set to null. Before returning from the execution of the method call, the post-condition is checked to ensure that the result is desired as expected in the type and the range of value.
4.2 Enforcing Security Contracts into Component Wrappers

4.2.1 Overview of Security Contracts

Security Contracts are defined as the mutual obligations for both collaborating sides to agree with. We use the constraints model tool to specify the additional security concerns and to enforce the concerns as policies on the desired component.

The Object Constraint Language (OCL) [14] is a formal language standardized by the OMG (Object Management Group) that allows the specification of constraints on MOF (Meta Object Facility) [13] or EMF (Eclipse Modeling Framework) [5] Ecore-based models. It is part of the UML (Unified Modeling Language) [15] which is also standardized by the OMG. In our work, we choose OCL as the language to specify the security constraint and implement it as security contract to weave into the component wrappers. The OCL is a query and constraint language which allows us to define the invariants, pre and post conditions. We use the implementation of the language, the widely used library Dresden OCL Toolkit [4], one of the software projects of the Software Technology Group at the Technische Universität Dresden, Germany, for our design of security contracts.

Applying contracts to component-based software is the best fit for quality assurance of interacting components. We take the example in the previous chapter to show the overview of security contracts for component wrappers. When the client sends the request to change the value of the temperature in Fahrenheit by \texttt{setF(double f)} method, the \texttt{TemperatureGUI} component expects that the parameter’s value should be in the valid range, for example, between 32 and 100 degrees Fahrenheit, otherwise
the request is not processed. Moreover, to ensure that the value is not changed by other parties, before returning the request result, it is also desired that the assigned value is the same saved in the TemperatureConverter component. Therefore, we have the described security policies implemented in OCL language shown in Figure 4.2.

```ocl
context TemperatureConverter::setF(f: double): void
  pre: f >= 32 and f <= 100
  post: TemperatureConverter.getF == f
```

Figure 4.2: Contracts Constraints between FarenheitGUI and TemperatureGUI components

The context specifies the scope of events where the constraints are imposed. The context also indicates all the invocations of method call `setF(double f)` on the component TemperatureConverter, it has the access to the parameter `(f: double)` from the context. The pre identifier specifies the pre-condition for verification before executing the method call, it requires that the parameter value should be within the allowed range, otherwise, the method call aborts and the exception events are written and recorded with the actual parameter values. The post identifier specifies the post-condition that should be met after the method call is finished. It ensures that after the value is set, the value retrieved from the component is exactly what was set by the method call, avoiding modifications from other sources at the same time. By applying security contracts for components, it increases the reliability of the interacting components, so that the errors would not be propagated through the system.

With the above constraints enforced, the behaviours of the method call are under observation. If there is abnormal behaviour which happens, such as the value beyond...
the range is passed to, the events are recorded and the action does not change the system at all, as shown below in Figure 4.3.

![Figure 4.3: Log Information with Contracts Enforced](image)

4.2.2 Process of Enforcing Contracts into Wrappers

Integrating security contracts into the components can increase reliability of components by automatic run-time verifications. However, the components are generally shipped without the source code. Inserting contracts into a component requires bytecode instrumentation or recompilation with available source code. The traditional aspect-oriented weaving mechanism compiles the aspect code and the original system’s source code together, generating a compiled code for the system, as shown in Figure 4.4 (a). The bytecode instrumentation without source code is also possible as shown in Figure 4.4 (b), but it requires expertise in low level bytecode manipulation. It involves complexities such as semantics check and correctness check at the bytecode level.

Although both mechanisms are supported and currently being researched, each mechanism has its limitations. The former approach, bytecode instrumentation involves operations on the bytecode program with expertise at the bytecode level, while on the other hand it is not realistic to acquire the source code of a third party component for recompilation in the latter approach.

When considering the pros and cons of both approaches of adding new features into
an existing system, we propose instrumenting new functionality into the wrappers, the protection layer of acquired components as shown in Figure 4.4 (c). Therefore, it not only avoids the necessity of manipulating the component’s bytecode with complexity, but also takes advantage of instrumentations by compilation with the component wrapper. In reality, the components are usually shipped without the source code, and it is difficult to operate on its byte codes. This makes the wrapping approach more realistic in practical use. A comparison of these approaches is illustrated in Figure 4.4.

We adopt the paradigm of design by contracts to enforce the run-time constraints against the component wrapper. Enforceable security policies are defined in the rule language independently from the platform language. The rule specifies the contracts
as the mutual obligations for the interacting components. The contracts are implemented in a separate module and weaved into the wrapper via aspects. In this way, it is easier to manipulate on the wrapper’s source code by weaving into additional concerns, while keeping the underlying components untouched.

Multiple security contracts can be specified and enforced into wrappers via the aspect technique. There are many examples where the multiple security constraints are required to impose in the system. For example, a client component Client is sending request to the server servlet PostServlet to return the portal for user to post feedback as shown in Figure 4.5.

![Figure 4.5: Example of Enforcing Behaviour Contract in Wrappers](image)

The PostServlet has only the functionality of directing the user’s request to the feedback page. It does not have any other concerns such as validating a user’s account, or verifying the user’s privileges. However, as the requirement could be different in systems, the extra concerns may need be added. Let us look at three example applications: System A, System B, and System C. In System A, the feedback form is available to all visitors including anonymous users and logged in users, then the
implementation of PostServlet can meet the requirement without changes. In System B however, when the servlet is used, the feedback form is only displayed when the user is logged in, thus requiring appropriate validations to be added in PostServlet. In our approach, we enforce the additional validations as security policies on the component, so that the component can be re-used without modifying its functionality. Lastly in System C, it is required that only those users with granted privileges are returned the feedback page.

By wrapping the underlying PostServlet into the wrapper PostServletWrapper, the original component remains untouched with the primary functionality, as the wrapper takes care of additional concerns such as access control as shown in Figure 4.6 and roles check as shown in Figure 4.7. The two contracts present the pre-condition checking before proceeding the execution of the method call. The similar implementation of post-condition checking can also be done if the post contract is required for the verification on the returned result.

```
public class LoginContract extends Contract{

    protected boolean preConditionChecking(HttpServletRequest request, HttpServletResponse response) {
        /* Check the user has logged in to return the feedback page */
        HttpSession s = request.getSession();
        String user = s.getAttribute("username");
        return (user != null);
    }
}
```

Figure 4.6: Login Contract for Component Wrapper

The LoginContract shown in Figure 4.6 presents the contract for login concern. Line 1 indicates that the login contract extends from the abstract class Contract, which defines all condition checking methods. Line 3 to line 9 implement the method
of pre-condition checking. In the condition checking, the contract retrieves the information from the current context, as is indicated from line 5 to line 7. It retrieves the current session object, and obtain the value of currently logged in user’s name. If the user is already signed in, it returns true value indicating that the execution can proceed because the user is a logged in user, and vice versa.

```java
public class AuthContract extends Contract{
    protected boolean preConditionChecking(HttpServletRequest request, HttpServletResponse response)
    {
        /* Check the user has granted privilege in to return the feedback page */
        String username = request.getContextParameter("account");
        if (grantedPrivilege(username))
            return true;
        else
            return false;
    }
}
```

Figure 4.7: Authorization Contract for Component Wrapper

The AuthContract shown in Figure 4.7 presents a different contract for access control concern. Line 1 and line 2 implement the condition checking methods similarly to LoginContract. Line 4 to line 10 implement the access control verifications. It firstly retrieves the information from the current request context, and obtains the value of logged account. The access control check is executed on the account by the system. If the privilege is granted by the system, the execution proceeds, and vice versa.

We design the security contracts using aspect-oriented programming technology, separating the component wrapper and the inserted contracts, thus avoiding the different enforced concerns hard coded. The contract is designed as a separate Java module, and is referenced in the required aspects, such as before, after, or around
the captured method calls. These join points are specified in the OCL language by
\texttt{pre}, \texttt{post}, and \texttt{inv} statements. The order of the contracts to be weaved follows the
order of the join points specified in the policy language. The implemented contracts
are then weaved into wrappers by defined aspects. When the wrapper intercepts
the method call, it evaluates the request information, as specified in aspects, and
invokes the corresponding contract implementation. The wrapper makes the decision
to forward the method call to invoke the underlying component if the contract checks
pass or if request parameters are sanitized, and to deny the request if the contract
checks fail. The run-time behaviours are managed and controlled within the wrapper.
The component stays behind the wrapper without access from external components
except for its associated wrapper.

4.3 Aspect Oriented Security Contracts

4.3.1 Architecture of Security Contracts in Aspects

The security contracts are specified as constraints around where the method call hap-
pens. There are many approaches of designing cross cutting concerns such as AspectJ.
The traditional aspect-oriented paradigm is to implement the non-functionality fea-
tures in the \texttt{aspect} modules including the \texttt{pointcut} which specifies the pattern of
location to be intercepted, the \texttt{joinpoint} which specifies the location to attach the
aspect code, and the \texttt{advice} which contains implementation of weaved concerns.
In our work, the aspect is used as the bridge between the target component wrapper and the enforced security contracts. The architecture of imposing security contracts as aspects into wrappers is illustrated in Figure 4.8. The Aspect module specifies the point cut indicating what to intercept, the join point indicating where to weave the additional code, and the advice, which is implemented as the contract. The Aspect does not contain the implementation of additional contract checking. It saves a reference to the contract module. For example, if there is a before advice is specified, when the specified point cut is satisfied, then the pre_condition functions in the contract is invoked in the context of before advice from within the Aspect module.

![Figure 4.8: Applying Contracts as Aspects into Component Wrappers](image)

The aspect module acts as the bridge between the join points and the constraints to be enforced. It specifies the pattern of intercepting calls, the location of attaching additional feature code, and the advice to be weaved into. To reduce the burden of the aspect module, it does not include the implementation of advice, vice versa, it retrieves the contracts from the contract factory to enforce, as shown in the diagram in Figure 4.9.
The aspect module contains all parts required by contracts instrumentation. The OCL Constraints contains the security policy specified in OCL (Object Constraints Language). It contains all required parts by <<Aspect>> module. The context in the OCL is interpreted as the pointcut in the <<Aspect>>. The <<JoinPoint>> module is specified by the statement of contracts such as pre, post, and inv statements. The join points can then be interpreted as before, after, and around advice. For each advice, the implementation of contracts is invoked from the <<Aspect>> module which has the reference to the <<Contract>> module. This separation increases flexibility and extensibility of enforcing multiple contracts at one joinpoint location.

The contracts are invoked by aspects in different stages: before the method call, after the return of method call, and around execution of the method call. The aspect module specifies the target method calls which the wrapper intends to observe. By weaving into wrappers the corresponding advice, the checking functions are performed at different stages. The events to capture are those sensitive method calls to the target component. These events indicate the occurred illegal requests or incompatible context parameters captured by the weaved security aspects. The mechanism of reporting these events are determined by the aspects, for example, logging the events, sanitizing the illegal parameters, throwing warning exceptions, and so on. The wrapper is bound with each wrapped component and manages incoming and outgoing requests from the component. The wrapper implements monitoring features with components registered as a reference, and impose security policies separately by weaving security aspects with reference to the implemented security contracts.
As the component-based software system is composed of different components cooperating with each other, the quality of the system is dependent to a great degree on how the components interact. From the functionality point of view, tight coupling between components leads to a high quality system, but from the security point of view, the loose coupling between components shows that the security of the system can be independent of individual components. Therefore, the security concerns are separated from the single component, and do not depend on individual ones and in case one component is compromised, the faults or malicious requests are not propagated to the other components.

4.3.2 Design of Security Contracts in Aspects

In order to separate the aspect and the contract module, we use the bridge pattern \[39\] to connect them for security enforcement. Figure 4.10 shows the implementation of the aspects and security contracts.
Figure 4.10: Enforcing Security Behaviour Contract with AOP

The Contract is an abstract class that contains the methods required for condition checking. It defines the different methods for each kind of condition checks for the captured method call, such as the pre-condition, post-condition, and invariant methods. All common contracts around the method call may be checked using a combination of these three condition checks. The SQLInjectionContract is the concrete contract class that implements the each condition checking method. The Contract object receives the current method call and its parameters that are passed in by the Aspect object.

The Aspect is an abstract class that specifies the process logic as shown in Figure 4.1. The targetPoint() defines the method that retrieves the point cut where the advice is imposed around. For example, the targetPoint() returns the before execution of a method which is invoked within a client component, then the aspect function defined in the method advice() will be executed when the pattern is satisfied with the pattern returned from targetPoint().
To summarize, within the function of the module, the security contract is designed and invoked from a separate module. The aspect specifies the context in which the security constraint is enforced between interactions of components and the rest of the system.

When another system reuses the module or extends the functionality, contracts may have to be changed accordingly. Instead of adding contract checks into corresponding components or classes, separating contract functionality from the implementation component is possible and is reasonable to be specified as a separate concern as a contract aspect.

Therefore, in our work we move the security contract functionality into the outer layer of the underlying component, the wrappers. The contract aware components lead to more secure products. It also separates the security contracts from the source code of the system. Thus it is a practical way to enable and disable the constraints of security policies against the underlying system via separating modules.

4.4 Integrating Contracts into Wrappers by Aspects

The monitoring framework consists of three major modules: Component Wrappers, Security Contracts, and Aspects. Component wrappers are used to enhance and confine the features of the underlying components. Incompatibilities are the typical problems when using components, thus requiring adaptations to comply with system specifications. Security Contracts are used to insert monitoring features into component wrappers at appropriate verification points, thus the run-time behaviours
are under observation by wrappers. Integration of two parts occurs when they are connected by the aspect bridge. The Aspect defines the pattern where and when to instrument additional monitoring code with security contract verification. The completed sketch diagram of the monitoring framework is illustrated in Figure 4.11.

![Figure 4.11: Monitoring Framework Integration with Wrapper and Security Contracts with Aspects](image)

The completed monitoring framework is glued together with the component wrappers, security contracts, and the aspects. The wrappers are used to register the association with the wrapped components. The wrapper acts as an invocation handler of the underlying component. The method calls on the underlying component are
intercepted by the wrapper and the wrapper makes the decision to forward or deny the request. The security contracts are used to verify the appropriate behaviours between the interacting components. The components must satisfy the mutual obligations specified by the contracts in order to collaborate with each other.

The aspect is attaching advice to multiple places across the modules of the system. The advice contains the reference to security contracts for run-time verification. The aspect module defines the pattern of the intended monitoring events, such as the method call, field properties access, and thrown exceptions. It evaluates the events on the underlying component.

The modularity of the monitoring framework offers the advantage of flexibility and extensibility:

**Flexibility**  The contracts are pluggable and enabled on demand. The aspect module defines only the pattern and location of instrumentation. Moreover, multiple security contracts can be instrumented into wrappers by customizing the aspect definitions.

**Extensibility**  By separating Wrappers, Contracts, and Aspect modules, it is easy to replace any of the modules. For example, instead of wrapping component interfaces, it may be required to enhance the underlying components with additional functionalities, then the wrapper can be sub-classed for extensibility. It may also require the audit contracts instead of security verification contracts to be instrumented, then the contracts can be replaced without changing the other two modules. In our approach, we specify the aspect around the method call to verify the run-time behaviours, however, developers can also define different patterns in aspects to instrument the
contract into wrappers. Replacing one module of the framework without changing the others offers great extensibility.

4.5 Summary

In this chapter, we illustrated an approach of specifying security contracts in OCL policy language, and presented the process of weaving security contracts into wrappers via aspects. We illustrated the flexible design of security contracts and the weaving mechanism with aspects and the complete monitoring framework was shown to prove its flexibility and extensibility.
Chapter 5

Experimental Evaluation

In the previous two chapters, we presented the design of component wrappers and enforceable aspect-oriented security contracts, and demonstrated the wrapper-based monitoring framework without touching the underlying components. In this chapter, we evaluate our proposed approach by monitoring SQL injection, Cross Site Scripting and Access Control policy in a component-based web application, which contains these vulnerabilities.

Section 5.1 gives the environment in which we conduct experiments including the target component platform, target vulnerabilities, target application, and the supporting tools; the experiments are illustrated in Section 5.2. We conduct a series of experiments with our monitoring approach on a component-based web application containing typical web vulnerabilities. We design the corresponding wrappers, aspects, and contracts to monitor typical security vulnerabilities, SQL Injection (Section 5.2.1), Cross Site Scripting (Section 5.2.2) and Access Control (Section 5.2.3). Section 5.3 summarizes this chapter.
5.1 Evaluation Environment

5.1.1 Target Platform

Java Enterprise Edition (JavaEE) is the representative example of component technology in Java language. The adoption of Java language to implement large-scale complex component-based systems eliminates the problems of direct memory access such as buffer overruns. However, despite Java's safety features, it is possible to make logical programming errors that lead to vulnerabilities such as SQL injections and cross site scripting attacks, and even the exploitation of weak access control.

Platform Java EE adopts MVC (Model, View, Control) pattern to implement web applications. It represents the components in three layers respectively. The View component is the presentation layer of the application, which is the display of information returned from server side. The Control layer receives the request from View component and return the computation results. The Model layer represents the data entities of the system.

MVC design pattern is widely used in enterprise applications. The application is divided into three separate layers, each implementing system functionality as a component module. We apply our monitoring framework to wrap modules in each of these three layers, using the current most popular component models in Java EE platform. We adopt our approach to investigate the security vulnerabilities between two modules, including XSS (Cross Site Scripting) issue between a View module and a Control module, SQL injection problem between Control and Model. We also consider access control to exposed interface which causes loss of information via unauthorized actions. We separate the security functionality from a component into its wrapper to
manage these security concerns.

5.1.2 Target Vulnerabilities

Many application server vendors have their own mechanisms to monitor and control transactions between the deployed components. This requires special annotations or XML descriptions, and such specific requirements are only applicable to that particular AS (Application Server). Our framework targets on the component as a Plain Old Java Object (POJO) that is applicable for all the application servers. We apply our framework to monitor the following security events: input validation including SQL injection and cross site scripting (XSS), and the access control concern.

SQL Injection

SQL injection is a popular problem of exploiting a database by injecting the SQL statements into the user input to break the SQL commands. The user input is received by the control layer (Java Servlet) and the query statement is constructed with the user input in the servlet handler. The model layer (data persistence) receives the query statement usually without security checking if the received statement is a valid query statement. If the user input data is not assessed before building the query statement, the malicious code can be easily injected into the query statement, and the database is then manipulated without passing the authentication. The additional input checking can be performed at either the view layer (web pages in browser) or the control layer (servlet). At the view layer, a JavaScript function can be used to perform real-time checking when the user is ready to submit the request, without sending the request to the control layer. In our work, we monitor this at the control
layer, to wrap the control handler to process the incoming request.

**Cross Site Scripting (XSS)**

The cross site scripting problems happen between the View component (JSP) and the Control component (Servlet) when request parameters contain invalid script elements. If it does not have sanitation on the parameters, the embedded script can lead to illegal behaviours that are executed in the browser. The input validations can be done either in the View component or in the Control component. In our approach, the illegal requests are intercepted and validated by wrappers, avoiding the reliance on other components to perform the sanitation task.

**Access Control**

Component modules are implemented with maximum re-usability which exposes as many interfaces as possible to be reused in multiple places in the system. A database access component is used by the Java developer to operate on databases with various methods. It receives the request from the servlet. As the interface is exposed public, it can also be exploited by other native clients. For example, the direct access of a desktop client application to the component, may be beyond the expectation of the component developer. Therefore, access control is required to be monitored for avoiding the unexpected access from suspicious clients.

**5.1.3 Target Example Application**

We conduct experiments on a flawed web application WebGoat [16]. The WebGoat application is a deliberate insecure web application used in purpose of demonstrating
the typical web security vulnerabilities and is implemented in the Java EE paradigm. The WebGoat application contains the security vulnerabilities such as SQL injection and cross site scripting. We design the corresponding wrappers, aspects, and contracts to monitor these events for the WebGoat application.

The WebGoat application is implemented in MVC design pattern. The architecture design of the application is shown in Figure 5.1.

![MVC Design Pattern in WebGoat Architecture](image)

**Figure 5.1: MVC Design Pattern in WebGoat Architecture**

The `HammerHead` is the servlet controller to delegate the user’s requests and forward the request to the JSP views. The controller delegates the requests to `ActionHandler`, which is a lesson object. The `ActionHandler` then executes the business logic, load the data into the WebGoat’s `WebSession` object, and then turn over the control to the presentation component, the JSP view pages. For tutorial purpose of presenting different security vulnerabilities, WebGoat application implements the lesson object as a component containing flawed business logics.
For example, as Figure 5.2 shows, when the user sends the request to view SQL injection results, the HammerHead controller dispatches requests to the lesson object SQLInjection, which includes the execution of SQL Infection scenarios. It registers the actions that contain the SQL Injection problems, such as the LoginAction and the ListProfile modules. The vulnerable logics in lesson objects are executed by the actions and the results are stored in the WebGoat’s WebSession object. The servlet controller also forwards the request to the presentation layer. The presentation layer is a JSP page consisting of web page layout templates and the lesson content from SQLInjection.jsp. The lesson content is generated from the current session, and thus presenting the unexpected results.

The ActionHandlers is the component of each lessons and implements all business logics that contain all the security problems, including SQL injections by interacting with databases, Cross Site Scripting problems by returning results to lesson’s JSP files. Therefore, our target component are these actions that implement various security
scenarios. We design the wrapper for each action component, and instrument aspect code to monitor the security events that occur in the scenarios.

5.1.4 Supporting Tools

Application Servers

To make it applicable to all component application servers, we deploy the instrumented wrappers with components in association with three representative application servers:

- Glass Fish, an open source applications by Sun Micro Systems [7].
- Open source JBoss Application Server, which has the rich AOP feature [10].
- JOnAS is an open source implementation of Java EE standard compliant application server. It was formerly developed and hosted by the ObjectWeb (OW) consortium, under the LGPL open-source license [12].

Development Environment

- Development Systems: Java EE 1.5, Servlet [8]. Servlet is a Java object on Java EE platform. It receives the request and generates the response based on the request. It has the access to the web container information.
- Deployment System: Tomcat 5.5 [1]. Tomcat is an open source web container used to deploy web applications.
- Integrated Development Environment: Eclipse 3.4.0 [6], Dresden OCL2 for
Eclipse [4], AspectJ [2]. Eclipse is a popular Integrated Development Environment (IDE). It supports the third-party plugins installed for OCL and AspectJ. Dresden OCL2 toolkit is a plugins OCL support in Eclipse. The AspectJ is an aspect-oriented language for Java.

5.2 Experiments

5.2.1 SQL Injection Monitoring

SQL injections are caused by unchecked input being passed to the back end database for query operations, leading to unintended consequences. It may also lead to the unauthorized access of the credential information.

In the SQLInjection component, we choose the Login action for monitoring. The Login action executes the business logics of user’s login to systems. The business flow of invoking component Login is shown in Figure 5.3. The HammerHead is the request handler component which receives all requests from the client. The SQLInjection component reads the result returned from the action component, and loads the result data into JSP files and sends these back to the client. The Login action component contains the main functions of the SQL injection issue. It retrieves the request data from the current session, which is obtained by its defined method HandleRequest. The method then passes the user input to login method in Login component to perform the login action.
CHAPTER 5. EXPERIMENTAL EVALUATION

Figure 5.3: Sequence Diagram of Executing SQL Injection Actions

**Specification**

The SQL injection problem is an issue caused by invalidated user input. User injects the SQL statements into the user input to taint the generated SQL execution. The injected statements contain SQL keywords that can affect the original SQL query execution and generates a new one that is harmful to the system.

The SQL injection happens when the user passes in the parameter with illegal and disallowed data. In a JSP page, the user inputs its `name` and submits to query the user’s information. The component `SQLInjection` lesson object executes registered actions related to SQL Injection scenarios, in our example, the `Login` action. The `Login` action receives the request, retrieves the parameters from the request, and constructs a SQL query operation based on the parameters. Figure 5.4 shows part of the functions in the `Login` action component. The method `handleRequest` is passed in with the current session object. The user input for user name and password
CHAPTER 5. EXPERIMENTAL EVALUATION

is retrieved from the session object. Line 9 invokes the method to login with the parameters of the current session, the user name, and the password. Line 13 to line 22 execute the login action with the passed user input. Line 16 constructs a query statement from the user name and the password that are passed in. Line 18 to line 22 create the connection to the database and performs the database query with the constructed query statement.

```java
public class Login extends DefaultLessonAction{
    // private properties
    // constructor of Login
    public void handleRequest(WebSession s) throws ParameterNotFoundException, ValidationException{
        // System.out.println("Login.handleRequest()");
        // retrieve employeId and password from WebSession
        // Attempt authentication
        boolean authenticated = login(s, employeId, password);
        // After login, update lesson status.
    }

    public boolean login(WebSession s, String userId, String password){
        // System.out.println("Logging in to lesson");
        // codes skipped here
        String query = "SELECT * FROM employee WHERE userid = " + userId + " and password = " + password + "'";
        Connection conn = WebSession.getConnection(s);
        Statement answer = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
        ResultSet answer.results = answer.executeQuery(query);
        // process the query result
    }
}
```

Figure 5.4: Scenario of SQL Injection in Login Action Component

As the example shows, we find that the string of `query` is a SQL statement that is generated from the parameter of `username` and `passwd`. If the illegal data such as the sensitive SQL keywords are injected into the parameter, the method call causes the SQL injection problem. For example, if the values of `username` and `passwd` are both
set to ' or 'a' = 'a', the query statement becomes:

```sql
SELECT * FROM employee WHERE userid = '' or 'a' = 'a' and password = '' or 'a' = 'a'
```

The expression of `userid = '' or 'a' = 'a'` is evaluated true as it evaluates to the value of `a' = 'a'`, which is always true. The same true value is evaluated for the password field. Therefore, by injecting SQL keywords into user input data, the attacker can easily bypass the authentications even if the attacker does not have the correct login information.

Monitoring the component’s interfaces can capture the data that is passed into the method calls. From the context of the passed data, the origin of the method call can be traced back. It is a way to block the flow of malicious data into the component.

**Wrapper Design**

We generate the wrapper for the `Login` component. According to the specification, the `Login` component has the following interfaces as shown in Figure 5.5. The information is generated with `javap` program to retrieve the available interfaces from the compiled class. Line 1 and line 2 indicate the class name and its extended class; line 3 to line 5 indicate its constructor and the required parameters; line 7 to line 13 show all available interfaces exposed for clients to request.
We choose the `login()` method to monitor the `Login` component in our work. The wrapper we designed is a `Login` action component with the enhanced `login()` method, as shown in Figure 5.5. We separate the implementation of target monitored method into the wrapper. In this way, the additional features can be inserted into the wrapper instead of touching the underlying component.

Line 1 declares that the wrapper extends from the `Login` component, and it also implements the interface `InvocationHandler`, which is an interface defining all proxy invocation methods. Line 2 is the underlying component instance saved as a property in the component wrapper, so that it can access its service from this instance. Line 4 to line 15 are the constructors of the wrapper. The parameters for the underlying component and the component instance itself are passed in for the construction of the wrapper. Line 17 to line 24 implement the binding of underlying components to the component wrapper. It returns a proxy object that will intercept the method call to the bound component, and the proxy object is able to delegate the method to...
any destined objects. Line 22 indicates that all interfaces available for the underlying component are under the observation of this proxy. The proxy is a private field of the component wrapper, and it is the actual delegate of the underlying component. Line 26 to line 30 implement the target `login` method with a simple redirection. The reason we implement it in this way is to avoid the modifications on the underlying component.

The aspect-oriented instrumentation actually enters the intercepted method call and instruments additional features in the intercepted method call. In our work, the additional instrumentation code is inserted around the redirection invocation of the underlying component, without entering the component itself, as shown in line 27 and line 29. Line 32 to line 44 implement the `invoke` method required by the implemented interface. It contains the logging information indicated in line 37 and line 43 to record the events of entering the method and exiting the method. Line 38 and line 39 mean that if the target method call is detected, the method call is redirected to the wrapper for invocation, which are shown in line 26 to line 30. Line 40 and line 41 forward the rest of the unmonitored method call directly to the underlying component, without much delay to avoid performance concerns.
```java
public class LoginWrapper extends Login implements InvocationHandler {
    private Login instance;

    public LoginWrapper(GoatHillsFinancial lesson, String lessonName, String actionName, LessonAction chainedAction) {
        super(lesson, lessonName, actionName, chainedAction);
        // TODO Auto-generated constructor stub
    }

    public LoginWrapper(GoatHillsFinancial lesson, String lessonName, String actionName, LessonAction chainedAction, Login original) {
        super(lesson, lessonName, actionName, chainedAction);
        this.instance = original;
        // TODO Auto-generated constructor stub
    }

    public Object bind(Object delegate, Object proxy) {
        this.instance = delegate;
        this.proxy = proxy;
        return Proxy.newProxyInstance(
            this.instance.getClass().getClassLoader(),
            this.instance.getClass().getInterfaces(),
            this);
    }

    public boolean login(WebSession s, String username, String passwd) {
        // before join points
        return instance.login(s, username, passwd); // execution join points
        // after join points
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // TODO Auto-generated method stub
        // logg the calling methods
        Logger.info(Level.INFO, "calling "+method.getName()+" methods ...*");
        if (method.getName().equals("login"))
            method.invoke(this, args);
        else
            method.invoke(instance, args);
        // logg the exiting methods
        Logger.info(Level.INFO, "exiting "+method.getName()+" methods ...*");
    }
}
```

Figure 5.6: Wrapper for Login Component
Security Constraints

The security constraints for SQL Injection is the validation on the user’s input information. We need to check that the input data does not contain any SQL keywords, such as “or”, “and”, and “union”. The pre-condition of the method call is that the parameter `username` should not contain those defined keywords. The specification of security constraints in OCL (Object Constraint Language) is shown in Figure 5.7.

```ocl
context LoginWrapper::login()
pre: SQLInjectionContract.preConditionChecking(args)
```

Figure 5.7: OCL specification for SQL Injection

It specifies the pattern where we impose the constraints, which is the method call of `login()` method in the context of component `LoginWrapper`. It defines the pre-condition that the parameters should be passed to the `preConditionSQLInjection()` method defined in the `SQLInjectionContract` component for sanitation. Here we can also add more preconditions as security constraints. Multiple constraints can be implemented in multiple contracts and further generate multiple aspects. The aspect code can be all applied at the appropriate join points.

Security Contracts

The security contracts are shown in Figure 5.8. The contract module is an independent object that implements the validation functionality. It does not have knowledge of the target it is applied to. So the contract can be applied to different interacting components, as required by users. The contract object passes an array of parameters, and it goes through all the parameters to detect if each element contains the sensitive
SQL statement keywords. The contract is implemented in this way for its flexibility and extensibility.

```java
public class SQLInjectionContract extends Contract{
    ArrayList keywords = new ArrayList(
        " and ", " or ", " union ", ";");

    public static boolean preConditionChecking(ArrayList parameters){
        for (String p : parameters) {
            for (String key : keywords){
                if (p.toLowerCase().indexOf(key) >= 0) {
                    log.warning("Parameter %s contains the sensitive keyword [%s]", p, key);
                    return false;
                }
            }
        }
        log.info("Pre-Conditions on Parameters Checking Passed.");
        return true;
    }
}
```

Figure 5.8: Security Contracts of Login SQL Injection

Line 1 declares that the concrete contract extends from the abstract contract class, which defines all condition checks methods. Line 2 to line 4 contain the list of illegal SQL keywords that are not allowed in the user input. The other possible keywords such as `join`, `sort`, and `if` can be added in the list. As in the WebGoat application, the other keywords have been covered at the client side, so here we only put the needed ones. Line 6 to line 18 implement the contract checking function. It checks all the received parameters to see if they contain any of the keywords in the list and returns the corresponding results indicating the value of the condition checking. The `postConditionChecking(args)` can also be implemented in the contract, as they are all defined from the abstract `Contract` class.
Aspect Code

The generated aspect code by OCL library is shown in Figure 5.9. It first retrieves the scope of the constraints, which is the target component, `LoginWrapper`, as in our example. The default pointcut for the constructor of the wrapper is generated so that we can instrument extra features when the wrapper component is constructed. The aspect specifies that before the `login()` method is executed, it passes the parameters to the contract module to check if it has injected data, so that the interface can be invoked safely.

Line 1 and line 2 indicate that the aspect is generated from the OCL policy language and the aspect is named with the word “Pre” to indicate that it is an aspect to instrument the before advice. Line 7 and line 8 define the point cut of the wrapper’s construction, so that the aspect goes into the wrapper instance. Line 13 and line 14 define the point cut of the `login()` method in the wrapper, which is actually a redirection to the `login()` method of the underlying component. Line 21 to line 27 implement the before() advice for the above defined point cuts. It then invokes the contract checks by passing the contract module the current context information as the `args` shown in line 22. If the contract check is passed, it proceeds to the invocation on the underlying component, as shown previously in Figure 5.6.

The aspect is designed as a bridge for the wrapper and the contract. It is not affected by the implementation changes of the wrapper and the contract.
5.2.2 Cross Site Scripting Monitoring

Cross Site Scripting (XSS) problem is another type of input validation issue for web applications. Different from the SQL Injection, the XSS injects the sensitive data in the output result to the client side. For example, after the user sends the request, the server responds by writing the HTML web pages to the user. In some scenarios, the web server generates the page file from the user’s input. However, the user’s input
may contain sensitive keywords for HTML file, for example, the `<script></script>` tags. When the web page is returned to the user, the maliciously injected scripts are executed in the browser, thus leaking the user’s privacies that are stored in cookies via `document.cookies`.

As an example, in WebGoat, a user searches and views an employee’s profile; when one malicious user injects the illegal script code in his profile, when other users, for example, the HR (Human Resource) people view this employee’s profile, the script code injected by the malicious user are executed, and the local data information are at risk and leaked to attackers by the injected scripting code. The process flow of this problem is shown in Figure 5.10.

![Figure 5.10: Process Flow of Cross Site Scripting Problem](image)

The attacker, who can be a regular user created in the system, logs into the system and updates his profile information with illegal scripting codes injected in the web page. The other users, who can be any one with view permissions to the web page, open the same page that was injected with the scripting codes. The scripting codes
are executed when the page is loaded. The local information stored in the browser cookies is sent out to the attacker who placed the code.

**Specification**

The XSS problems exist due to insufficient validations of the posted data by end users. The vulnerable web application does not consider the necessities of data validations. However, as this input information is used as the view of data model displayed to the end users, thus causing executable code running right in front of the users.

In WebGoat application, the XSS problem exists when user views the other employee’s profile information. As illustrated in Figure [5.10](#), the web components are generated on the user’s input, for example, the user’s name field. If the text field does not have input validation, the generated web page content is at risk of being injected with a scripting code. In WebGoat, the web page consists of the user’s profile information. The data model is implemented by the **Employee** component.

```html
1  <b>Username</b>
2  Username: &lt;br&gt; &lt;input type="text" name="username"&gt;&lt;br&gt;
3  Password: &lt;br&gt; &lt;input type="password" name="password"&gt;&lt;br&gt;
4  &lt;input type="submit" value="Login"&gt;
5    onclick="var xssImg=new Image();
6    xssImg.src='http://yehg.org:8080/WebGoat/catcher?PROPERTY=yes&u=' +
7      this.form.username.value + '&p=' +
8      this.form.password.value;">
```

Figure 5.11: Cross Site Scripting Code

When the user updates the profile, for example, the **firstname** field, the field should not contain any scripting code. An example of such scripting code is shown in Figure [5.11](#). It injects the HTML tag data into the input fields. It contains two input fields of **username** and **password**, and a login button. Line 4 to line 8 define
an action when the login button is pressed. It retrieves the value from `username` and `password`, and sends it to another website which has the handler to process the received information. If this code is injected successfully, the other users will see the injected page as shown in Figure 5.12 when they open the profile page. A user will mistakenly enter his or her personal information in the username and password fields and press the login button, which will send the entered personal information to the attacker.

Each user can update its own profile information, and the profile is public for other employees to view. Each text field is the text value to the property field of the `Employee` component via the method `setName()`. When the other employees view the profile page, it retrieves the profile information from the `Employee` component, however, the value to each fields are not validated and passed in the web page directly via the method `getName()` and thus causing it executed as the scripting code.

![Figure 5.12: Web Page Generated by Cross Site Scripting](image-url)

(a) The normal page without injection

(b) The injected page

Figure 5.12: Web Page Generated by Cross Site Scripting
CHAPTER 5. EXPERIMENTAL EVALUATION

The injected code can also steal the user’s private information by sending a `document.cookie` to the attacker. Therefore, the wrapper and aspect generation should target the actions against the `Employee` component.

**Wrapper Design**

The component `Employee` was implemented as a data model that can also be reused in different systems. No constraints are imposed on the data model, due to its modularity. However, it can be abused by attackers if the system that adopts the component does not provide additional concerns on this model, the users’ private information would be in leakage. Thus, we wrap the `Employee` component without touching the original component. The wrapper is shown in Figure 5.13 using an adapter design pattern.

We wrap all the setter and getter method calls, and verify the parameters passed in to assure that they are safe for generating the output to web pages. We omitted the overridden setter and getter methods in the figure to save page space. Line 1 to line 8 are similar to the previous example, declaring the extension from the underlying component, and construct the wrapper with the underlying component passed as the parameter. Line 9 to line 16 return the proxy object that intercepts the method call to the underlying component. Line 19 to line 31 implement the redirection of the intercepted method call. If the target method call is detected, it is directed to the wrapper for condition checking before forwarding to the underlying component as shown in line 15. The rest of the captured method calls are directly forwarded to the underlying component shown in line 29. The logging is also included to record the events of the method invocation on line 24 and line 27.
public class EmployeeWrapper extends Employee implements InvocationHandler{
    private Object instance;
    private Object proxy;

    public EmployeeWrapper(Employee original){
        this.instance = original;
    }

    public Object bind(Object delegate, Object proxy){
        this.instance = delegate;
        this.proxy = proxy;
        return Proxy.newProxyInstance(
            this.instance.getClass().getClassLoader(),
            this.instance.getClass().getInterfaces(),
            this);
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        // TODO Auto-generated method stub
        if (method.getName().indexOf("get")>0 or method.getName().indexOf("set")>0 ){
            // logg the setter and getter methods
            Logger.info(Level.INFO, "calling " + method.getName() + " methods ...");
            method.invoke(this, args);
            // logg the setter and getter methods
            Logger.info(Level.INFO, "exiting " + method.getName() + " methods ...");
        } else
            method.invoke(instance, args);
        return null;
    }
}

Figure 5.13: Wrapper Component for XSS Component

Security Constraints

When the setter and getter methods are called, the security constraints are imposed to verify that the passed parameters contain scripting codes. Thus, in the same pattern as the previous examples, we check the contracts by passing the intercepted parameters to the XSSContract module, as illustrated in the next section.
The parameters of each setter and getter method should not contain scripting codes.

\begin{verbatim}
context EmployeeWrapper::get.*(String args) or EmployeeWrapper::set.*(String[] args)
pre: forall (methods(args) in context)(XSSContract.preConditionChecking(args))
\end{verbatim}

Figure 5.14: OCL Constraints for XSS Component

Line 2 indicates the context pattern to capture. It means that all the getter methods and setter methods in the `EmployeeWrapper` object should be matched. Line 3 indicates that for each method matched in the context pattern, the pre-condition check should be performed against the method call. The wild cards used in the context pattern provides a flexible way to match multiple method calls with similar signature patterns.

**Security Contracts**

Separated from the wrapper and the aspect, the detailed contracts are implemented as an independent module as shown in Figure 5.15. It also prepares the list of illegal keywords in scripting language, as shown in line 2. The other possible keywords such as `javascript`, `src`, and the `function` can be added depending on the context that is assessed. In our example, we only target the special script tags. Line 4 to line 12 implement the parameter checking against the list of those keywords. The more complex evaluation can be added to the contract module without depending too much on the wrapper or the aspect.
CHAPTER 5. EXPERIMENTAL EVALUATION

public class XSSContracts extends Contracts{
    ArrayList keywords = new ArrayList("<", ">");

    public static boolean preConditionChecking(String[] parameters){
        for (String p : parameters) {
            for (String key : keywords) {
                if (p.toLowerCase().indexOf(">") >= 0 || p.toLowerCase().indexOf("<") >= 0) {
                    log.warning("Parameter %s contains the sensitive keyword [%s]", p, key);
                    return false;
                }
            }
        }

        log.info("Pre-Conditions on Parameters Checking Passed.");
        return true;
    }
}

Figure 5.15: Security Contracts for XSS Component

Aspect code

The aspect code is generated to bridge the wrapper module and the contract module. The constraints specified in OCL are interpreted by the aspect generator provided in the OCL tool kit. It executes the contract code at the positions specified by the aspect module. The generated code is shown in Figure 5.16.

Line 3 and line 4 define the point cut for the target method call. In this example, we specify all the setter methods to monitor. Line 4 matches any methods starting with set name in EmployeeWrapper component that returns any type of result. Line 6 to line 10 implement the invocation of contract checking for the defined point cut. When the point cut is matched, the context of the method call such as the class name, and the parameter list are passed into the contract module for assessment. If it passes the contract checking, it proceeds to the invocation on the underlying component as shown in Figure 5.13.
5.2.3 Role Based Access Control Monitoring

Specification

Monitoring of secure access concerns involves the identification of incoming requests. The identity of components involved in the transaction is assessed, which can be named Component ID. Each transaction should be performed between two verified components. Any transactions between two non-verified components are considered as abnormalities. The request path verification is performed by the wrapper’s weaved security contracts, to check if the incoming request is from a valid component as specified in software architecture. The path knowledge with software architecture is defined in the monitoring knowledge base.

The “context” of the request path should also be taken into consideration. The “context” is the environment in which one session goes through, from the initial point to the component in which the secure request happened. The interactions together with the context will be observed in run-time and validate its health to check the
violations of unauthorized access. The “context” concerns will be weaved into the target system by aspect-oriented programming. The aspects are designed based on the “context” path shape analysis. The state of the requesting component should be monitored. We inject the verification code into the monitored wrappers to observe the secure state of the component invoking the operations of the others, so that the neighbours of that component would be notified of the problem once it occurs. The neighbouring component would consider it as a compromised component containing a malicious request, and will refuse the request. In this way, the data asset and information can be protected from any possible potential risks.

While the access control might be appropriately implemented, the user request injected with a malicious code might take control of root privilege and the system may process the request without suspecting its real identity. The track of the logic is necessarily taken into consideration in run-time monitoring. Therefore, the source of the request should be validated with the trace-back path of the request flow. Security problems could be traced to the fact that all of the COTS components were developed with lower security requirements than the composite system. The developers had failed to ensure the security of the system independent of the security of the individual components.

Wrapper Design

In WebGoat, we take the same example as in the previous experiment, when one employee A happens to hack into employee B’s profile page, there is an option of delete profile shown on the page. This option calls the doDeleteProfile in component DeleteProfile. When the component was developed, no security checks are
concerned. As shown in Figure 5.17, it would be ideal that an additional security constraint can be imposed on the call. Now since the interface is exposed, the necessary access control is desired to impose on the method call. Firstly, we wrap the component and filter the doDeleteProfile method. The wrapper is shown in Figure 5.18. When the doDeleteProfile method is invoked, it delegates to the wrapper first before forwarding to the underlying component.

![Diagram](image)

Figure 5.17: Adding Additional Access Control on Interface

The Figure 5.18 shows the wrapper for the DeleteProfile action component. Line 1 to line 7 define the declaration of the properties, the constructor and the setting values to the passed in parameter, which is the underlying component instance. Line 9 to line 13 implement the doDeleteProfile method, the before and after join points can be added with the advice around the direction to the invocation on the underlying component instance. Line 15 to line 22 return the proxy object that intercepts the method call to the underlying component. All interfaces are under observation as is indicated in line 20. Line 25 to line 35 implement the detection of the captured method call. If the target method call is detected, it is directed to the wrapper instance for invocation as shown in line 31 and line 9 to line 13. The rest of the
method calls are forwarded to the underlying component directly, indicated in line 33. The logging solution is also included at entering and exiting the method call, as indicated in line 29 and line 35.

```java
public class DeleteProfileWrapper extends DeleteProfile implements InvocationHandler{
    private DeleteProfile instance;
    private Object proxy;

    public DeleteProfileWrapper(DeleteProfile original) {
        this.instance = original;
    }

    public boolean doDeleteProfile(WebSession s, int userId, int employeeId) {
        // before join points
        return instance.doDeleteProfile(s, userId, employeeId); // execution join points
        // after join points
    }

    public Object bind(Object delegate, Object proxy) {
        this.instance = delegate;
        this.proxy = proxy;
        return Proxy.newProxyInstance(
            this.instance.getClass().getClassLoader(),
            this.instance.getClass().getInterfaces(),
            this);
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // TODO Auto-generated method stub
        // log the calling methods
        Logger.info(Level.INFO, "calling " + method.getName() + " methods ...");
        if (method.getName().equals("doDeleteProfile"))
            method.invoke(this, args);
        else
            method.invoke(instance, args);
        // log the exiting methods
        Logger.info(Level.INFO, "exiting " + method.getName() + " methods ...");
    }
}
```

Figure 5.18: Wrapper Component for RBAC Component
Security Constraints

Before forwarding the deletion action calls to the underlying component, we define the security constraints that the action should be proceeded under the condition that it satisfies the access control condition. The constraints are shown in Figure 5.19. It defines the precondition that it should pass the condition checking defined by RBACContract module.

```ocl
context DeleteProfileWrapper::doDeleteProfile(WebSession s, String[] args)
pre: RBACContract.preConditionChecking(s, args)
```

Figure 5.19: OCL Constraints for RBAC Component

Line 1 specifies the context pattern for the captured method call. It defines that the `doDeleteProfile` method in the `DeleteProfileWrapper` component should be matched, the parameters are the current session object indicated by `WebSession`, and the rest of the account information indicated by the `args`. Line 2 indicates that if the above specified context pattern is matched, the corresponding context information is sent to the `RBACContract` to perform the pre-condition check.

Security Contracts

The `RBACContract` module defines the precondition checking that validates that the user should be in authorization status to perform the action on the target object, as shown in Figure 5.20. When the method `doDeleteProfile` is invoked, it will first check the condition that if the current user’s ID is in an authorized status by a system defined access control list. Thus, the access control is imposed on the method calls where needed without any modifications on the underlying components.
public class RBACContract extends Contract {
    private int userId;
    private int targetId;
    private WebSession s;

    public static boolean preConditionChecking(WebSession s, String[] args) {
        this.s = s;
        this.userId = args[0];
        this.targetId = args[0];
        if (s.isAuthorizedInLesson(userId, RoleBasedAccessControl.DELETEPROFILE_ACTION, targetId)) {
            log.info("Pre-Conditions on Parameters Checking Passed.");
            return true;
        } else {
            log.warning(userId + " has no authorization to perform current action: " + RoleBasedAccessControl.DELETEPROFILE_ACTION + " ON " + targetId);
            return false;
        }
    }
}

Figure 5.20: Security Contracts for RBAC Component

This contract accepts an additional parameter, which is the WebSession object. The reason is that the access control needs the session information to track the source origin of the method call where it was initiated. Line 2 to line 4 define the caller, callee and the involved web session. Line 6 to line 18 implement the verification of the permissions of the caller on the callee. It invokes the privilege verification function in the WebGoat application, passing into it with the caller ID, the callee ID (which is also the target ID), and the action the caller is performed against the callee (which is the delete action as is indicated in line 10). Line 10 invokes the function that the WebGoat application uses to verify its role-based access control. We use it here to complement access control on the other method calls that were not covered when building the system.
Aspect code

The aspect code is deployed in the system, together with the contracts, and wrappers. The method call to deletion action will be constrained by access control check. A user is also able to define different mechanism of access control like role based access control, to adapt to the system security requirements. The generated aspect code is shown in Figure 5.21.

```java
@Generated
public privileged aspect PreAspect{
    protected pointcut allDeleteProfileWrapperConstructors(org.owasp.webgoat.lessons.RoleBasedAccessControl.DeleteProfileWrapper aClass):
        execution(org.owasp.webgoat.lessons.RoleBasedAccessControl.DeleteProfileWrapper.new(..)) && this(aClass);

    protected pointcut alldoDeleteProfiles(org.owasp.webgoat.lessons.RoleBasedAccessControl.DeleteProfileWrapper aClass, WebSession s, String[] args) :
        execution(∗org.owasp.webgoat.lessons.RoleBasedAccessControl.DeleteProfileWrapper.doDeleteProfile(s, args)) && this(aClass);

    before(org.owasp.webgoat.lessons.RoleBasedAccessControl.DeleteProfileWrapper aClass, WebSession s, String[] args) : allDeleteProfileWrapperConstructors(aClass) | alldoDeleteProfile (aClass, s, args) {
        if (!RBACContract.preConditionChecking(s, args)) {
            // TODO Auto-generated code executed when constraint is violated.
            throw new RuntimeException("Error: Constraint was violated.");
        }
        // no else.
    }
}
```

Figure 5.21: Security Aspects for RBAC Component

Line 3 to line 7 define the two point cuts for instrumentation. The point cut in line 3 indicates the construction of the component wrapper. It is used for latter one to track the original source of the method call, and to check if the method call is requested from a legal component. The identity of the component on which
the method call is requested is stored in the session object as indicated by the parameter `WebSession` in line 6. Line 6 defines another point cut to match all the `doDeleteProfiles` methods. It means any kind of request to the `doDeleteProfiles` from the component wrapper `DeleteProfileWrapper`. The current session and the method parameters are passed together to the aspect. Line 9 to line 14 implement the advice for the above point cuts. The invocation to the contract checking is executed and the current context information such as the session object and the method arguments are sent to the contract module for condition checking. Therefore, the execution can proceed if the contract checking passes, which means the caller has been granted the permission to perform the method call. Otherwise, the exception will be thrown to the wrapper, and the execution will be stopped, thus stopping the suspicious attack breaking into the underlying component.

5.3 Summary

We have presented the implementation of our monitoring framework using aspect-oriented wrappers with contracts. The framework is applied to monitor security issues of component-based systems, SQL injection, cross site scripting, and access control problems.

From the evaluations and experimental results, we can see that it is flexible to enforce customized security policies on target components. The framework is applicable to components with standard specifications, without reliance on a particular application server environment. The flexibility is inherent in the monitoring framework as wrappers, aspects, and contracts are developed separately.
Chapter 6

Conclusion and Future Work

6.1 Conclusion

Component-based software is a popular technology in the software development area, because it reduces the cost of designing the system from scratch to a great degree. The implemented component software can be reused within multiple systems with the same interfaces, as long as they are compatible with the system specifications. While the component-based software brings great benefits with regards to re-usability, the quality of the acquired components is not guaranteed as it may come from various vendors to form the system. This leads to the necessity of the monitoring of the components. As an interface is a functionality access point for clients, we propose an approach to wrap the interfaces and weave around the interfaces to monitor the run-time behaviours of the method calls of the component.

In this thesis work, we presented the approach of monitoring the components using wrappers with contracts and aspects. We designed the wrappers for component monitoring, and presented an example application to add monitoring features into the
CHAPTER 6. CONCLUSION AND FUTURE WORK

component system via wrappers. We extended the functions from the underlying components, and generated the wrappers implementing the proxy features. The wrapper is deployed into the system to intercept the method calls of the wrapped component. We demonstrated the process of run-time verifications in component wrappers and provided the detail implementation of the component wrappers with UML diagrams. We designed contracts in OCL policy language and showed the aspect code generated from the security contracts. We designed the aspects that incorporate the security contracts as the reference. The process of enforcing the contracts with aspects into the wrappers is demonstrated together with the architecture of the integration solution. In the integration solution, we illustrated how the contract is executed via aspects in component wrappers.

Using our approach, we conducted experiments on a Java EE web application, which contains various vulnerabilities, such as SQL Injection, Cross Site Scripting problems, and the access control concerns. From these experiments, we found that it is convenient and flexible to design the monitoring features separately and implement them in independent modules, leading to extensibility and flexibility. The wrappers, aspects, and the contracts are implemented independently in separate modules from each other, which means, it is convenient to apply the same contracts to different components, or around different aspects, increasing the flexibility.

6.2 Limitations and Future Work

Our work is currently only applicable to method calls monitoring, as the interface is the only way we access component’s provided services. The internal behaviours of components also need to be monitored. The overhead and performance concerns
are not yet considered in the work. The current approach is targeted for a feasible and flexible way to deploy the monitoring features into the existing system. We believe that the major extra overhead incurs at the development time rather than the run-time.

The approach can possibly be extended in a couple of ways. In our work, the system has to be re-deployed with the wrapper, aspect, and contract modules. In future, it may be enhanced with a solution that the modules can be hot-plugged and played into the system, so that it will not affect the running system and no re-deployment will be required. On the other hand, more internal information of a component can be associated with the component as its metadata, so that the inspection of the component can return various properties of the component by rich API methods on those metadata. It will make the monitoring of the components easier in the future, while it will involve more work on unifying the standards for all existing component specifications.
Bibliography


[7] *GlassFish Community.* [https://glassfish.dev.java.net/](https://glassfish.dev.java.net/)


[39] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 2002.


