Graduate School Form
30 Updated 12/26/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Mehdi Azarmi

Entitled
End-to-End Security in Service-Oriented Architecture

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Bharat Bhargava Chair	
Hubert E. Dunsmore	
Vernon J. Rego	
Xiangyu Zhang	

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Bharat Bhargava

Approved by: Sunil Prabhakar / William J. Gorman 04/14/2016
Head of the Departmental Graduate Program Date

END-TO-END SECURITY IN SERVICE-ORIENTED ARCHITECTURE

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Mehdi Azarmi

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2016

Purdue University

West Lafayette, Indiana

To my wife and my parents

ACKNOWLEDGMENTS

First, I would like to thank my Ph.D. advisor, Professor Bharat Bhargava for his help throughout my Ph.D. In particular, he provided me freedom to pursue multiple research interests and he gave me a better understanding of the research methodology. I am also grateful to my committee members, Professor Xiangyu Zhang, and Professor Vernon Rego for serving on my Ph.D. committee and providing useful comments that helped me to think more deeply about my research. I also thank Air Force Research Laboratory (AFRL) and Northrop Grumman (Cybersecurity Research Consortium) for partially support my research. I am also grateful to Pelin Angin and other lab members for their collaborations and insightful discussions. Many thanks to my wife and my family for their continued support during my Ph.D. study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ASSC	Adaptive and Secure Service Composition
PME	Policy Monitoring and Enforcement component
QoS	Quality of Service
REST	Representative State Transfer
SLA	Service Level Agreement
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
TM	Trust Manager
TTP	Trusted Third-Party
UDDI	Universal Description Discovery and Integration
WS-*	Prefix Indicates web service specifications
WSDL	Web Services Description Language
XACML	eXtensible Access Control Markup Language
XML	eXtensible Markup Language

ABSTRACT

Azarmi, Mehdi Ph.D., Purdue University, May 2016. End-to-End Security in Service-Oriented Architecture. Major Professor: Bharat Bhargava.

A service-oriented architecture (SOA)-based application is composed of a number of distributed and loosely-coupled web services, which are orchestrated to accomplish a more complex functionality. Any of these web services is able to invoke other web services to offload part of its functionality. The main security challenge in SOA is that we cannot trust the participating web services in a service composition to behave as expected all the time. In addition, the chain of services involved in an end-to-end service invocation may not be visible to the clients. As a result, any violation of client's policies could remain undetected. To address these challenges in SOA, we proposed the following contributions. First, we devised two composite trust schemes by using graph abstraction to quantitatively maintain the trust levels of different services. The composite trust values are based on feedbacks from the actual execution of services, and the structure of the SOA application. To maintain the dynamic trust, we designed the trust manager, which is a trusted-third party service. Second, we developed an end-to-end inter-service policy monitoring and enforcement framework (PME framework), which is able to dynamically inspect the interactions between services at runtime and react to the potentially malicious activities according to the client's policies. Third, we designed an intra-service policy monitoring and enforcement framework based on a taint analysis mechanism to monitor the information flow within services and prevent information disclosure incidents. Fourth, we proposed an adaptive and secure service composition engine (ASSC), which takes advantage of an efficient heuristic algorithm to generate optimal service compositions in SOA. The service compositions generated by ASSC maximize the trustworthiness of the selected services while meeting the predefined QoS constraints. Finally, we have extensively studied the correctness and performance of the proposed security measures based on a realistic SOA case study. All experimental studies validated the practicality and effectiveness of the presented solutions.

1 INTRODUCTION

Service-Oriented Architecture (SOA) is a software architecture pattern that promotes composition of an application from a number of loosely-coupled services. Each of these services provides a specific functionality and is governed by a service provider. The main goals of SOA are reusability, interoperability, and scalability. In this chapter, we first discuss motivation behind this dissertation, which targets the security of the SOA. Second, we formally define the problem statement. Next, we discuss the security requirements in SOA and the design principles. We follow by briefly describing the contributions and scope of this research work. We conclude this chapter by providing the organization of this dissertation.

1.1 Motivation

With the rapid growth of enterprise web-based services and growing popularity of cloud computing and microservices, SOA is becoming a key software paradigm for the development of distributed services and applications. Considering such a rapid growth of cloud computing as an enabling technology, SOA has shown to have a large market share in the enterprise software world. For example, Zdnet [1] reports that SOA market grows 17% a year and it is expected to reach \$10 billions by 2015. SOA promotes service reusability through composition of services. Therefore, heterogeneous services within an organization or across multiple organizations can be chained together to provide a more complex functionality. Integration among enterprise applications has been among the most challenging problems in the software engineering in the past decade. SOA is an architectural style, which enables and simplifies the interconnection between heterogeneous applications inside or across organizational boundaries.

Security is a major challenge in SOA. Even though a significant amount of effort has been done to study and improve the security of web services, still, majority of the proposed solutions address the security challenges of single services or client-server models. In fact they fail to address the multi-hop nature of service invocations that happen in SOA. Current security solutions enable a service to control the access to its local resources or interaction with an immediate service it interacts with. SOA opens the door into new security challenges that are not present in the traditional client-server architecture. These new challenges are caused by multi-hop nature of SOA, which is composed of a chain of multiple services from independent service providers. On the other hand, because of the SOA architecture's open nature, it is not possible to directly access and change the third-party services in a SOA. Therefore, some services down in the invocation chain may access a malicious service without being detected.

For example, consider a chain of services S_1, S_2, and S_3, that service S_1 from company A accesses S_2, which is hosted in company B. And S_2, in turn, accesses s_3 from company C. In this scenario, s_1 (based on its negotiated policy) expects service s_2 to perform the requested functionality by itself or by using another trusted service (e.g., service s_4 from company D). However, for some reasons (potentially, with financial motives), s_2 decides to use s_3, which is not trusted by client. In such cases, client has no way to know that its sensitive information are exposed to company C. Similarly, company B may be hacked and the compromised s_2 sends the sensitive information to an unknown server. These scenarios, allow flow of private information along the chain of service invocation, which leads to information leakage. In such scenarios, service consumers have no visibility or control over untrusted service calls. Even though commonly used security protocols (such as SSL/TLS) can help to provide a point-to-point security, however, the lack of dynamic trust and end-to-end security makes it necessary to design a new monitoring and enforcement framework, which is able to verify the service invocations at runtime and ensure that service consumers' policies are enforced.

1.2 SOA Security Challenges

In this section we investigate the main security challenges in SOA and we discuss why the end-to-end security in SOA is important. these characteristics may lead to inclusion of untrusted services from unknown administrative domains.

SOA architecture is inherently open, decentralized, and heterogeneous, which encourages independent service providers to provide reusable and composable services. The lack of control over all participating services in such scenarios, makes it impossible to provide any kind of end-to-end security guarantee. These characteristics lead to a larger attack surface (compared to other traditional architectures), which magnifies the importance of protecting the data flows at all times.

The current trend of Internet shows that different service providers provide a diverse range of services, which are dependent on other services and service clients invoke those services dynamically without an adequate knowledge of the dependencies of those services. Moreover, the corporate service consumers are likely to compose new services based on the available off-the-shelf services and the required attributes (e.g. cost, delay, and availability). Therefore, these composed services depend on multiple service providers, and the final client may not have any kind of security or QoS guarantee. *End-to-end security* is a mechanism, which enables the service consumers to monitor or control the flow of information and service invocation chains from the initiation of service request until the reception of service response. Using this mechanism, service consumer will observe the potential breach of information and can identify what service is responsible for invocation of an untrusted third-party service.

We identify the following important challenges for the *end-to-end security in SOA*:

- *Lack of global visibility, accountability, and control.* In current SOA scenarios, users do not have a complete visibility or control over external service invocations within a service composition. In SOA applications, whenever a service subcontracts or outsources a subset of its functionalities to untrusted third-party services, the malicious invocations will not be visible by users and the malicious service will not be held accountable.

- *Multiple administrative domains.* SOA applications are usually composed of services from multiple independent providers. There are various stakeholders in SOA, including: service provider, cloud service provider, service consumers, and sometimes, federal agencies. Since administrative domains have different interests and capabilities, providing a uniform security solution across organizational boundaries is challenging and often leads to various inconsistencies and other security challenges. Most of the current solutions focus on the service provider policies. However, service consumers or other stakeholders may demand different levels of security depending on the circumstances.
- *No verification and validation of external services.* The execution of services are is not verified dynamically. Therefore, service consumers must select a service with no security awareness or only based on static trust assignments. Mission critical services cannot just trust other services without verification.
- *The security of SOA is not the same as security of web services.* New cryptographic protocol won't help by itself and current security measures (WS-* standards) are not enough. The widespread transport-level protocols (e.g., SSL and TLS), are not be responsible for the correctness and security of data after delivery to the target endpoint. Therefore, a malicious or misbehaving service cannot be detected with these protocols.
- *Trust mechanisms are not transitive.* Businesses place a lot of trust in their partners. However, a service, which is trusted by a business partner, may not be trustworthy to the service consumer.
- *Services change continuously and frequently.* Modern SOA applications are constantly evolving in response to the fast-changing business requirements. This characteristic leads to constant changes in the services (short product release cycles). Moreover, modern SOA systems are complex and large-scale. Therefore, any viable security solutions must capture the *dynamics* of these complex

systems and react to them properly. The existence of a platform and technology agnostic monitoring system is crucial to the security of SOA systems. Considering such rapid changes in services, a trusted service may lose its trustworthiness in a short period of time. Therefore, the security framework requires being adaptable and extensible. These requirements rule out most of the security solutions that are based on service certification.

- *SOA applications suffer from complex attacks and bugs (software faults, misconfiguration).* Modern SOA services are complex and far from being bug-free. any configuration error may lead to inconsistencies and disclosure of private information. As a result, the traditional security paradigm that uses a protection perimeter, which prevents outsiders from accessing organization systems, is not valid anymore in such systems. Such misbehaviors are the results of the collaboration between companies, which forces each company to share its data with its partners outside of its protection perimeter. Moreover, services in SOA may misbehave arbitrarily. This source of misbehavior is due to malicious activities of services, which may be the result of an attack or the service provider is malicious (services may get compromised). In these scenarios, malicious or compromised services make unauthorized external service invocations, which leads to data leakage and compromise in user's privacy.

1.3 Thesis Statement

In this dissertation, we propose a framework for end-to-end policy monitoring and enforcement in SOA.

The thesis statement of this dissertation is as follows:

Composite trust models backed by service execution monitoring, provide an effective framework for end-to-end visibility, accountability, policy monitoring, and policy enforcement in SOA. Such framework enables SOA consumers to detect and prevent *data disclosures in chain of services and adapt the SOA application to increase the overall security.*

1.4 Contributions

The SOA security challenges, presented in section 1, raise the need for a set of mechanisms and a framework for end-to-end security monitoring and enforcement in SOA. In particular, this dissertation provides the following contributions:

An end-to-end security architecture for SOA. (Chapter 2)
Most of the recent research work are either addressing the SOA security issues individually or they are technology-dependent (specific to SOAP-based or RESTful services or targeted to a specific composition technology such as BPEL). We designed a new security architecture which is technology-agnostic, non-intrusive (no need to change services), and source-code independent (no need to access or change the source code of target services). This architecture takes advantage of aspect-oriented programming (AOP) that is highly extensible and flexible.

The proposed security architecture is composed of two main components: trust manager service (*TM*) and policy monitoring and enforcement component (*PME*). The TM is itself composed of multiple components including: trust engine (*TE*), session management engine, adaptive secure service composer (*ASSC*), and policy engine (*PE*). The PME component is also composed of two parts: inter-service and intra-service (taint analysis) monitoring and enforcement components. The interaction between these components enables the end-to-end monitoring and enforcement of user-defined security policies. This goal is achieved through tracking the global and session-based composite trust using service execution monitoring. In chapter 2, we discuss how all components of the system are fit together to provide a holistic security framework for end-to-end security in SOA.

New composite trust models for SOA. (Chapter 3)
We designed a new dynamic trust management service for SOA that maintains the

trustworthiness of services quantitatively, based on the actual execution and behavior of those services. In this chapter, we propose and design multiple composite trust strategies and algorithms based on graph abstraction for SOA. The main components of this system are trust engine, which is responsible for calculating and maintaining composite trust, and session management engine, which is responsible for maintaining end-to-end sessions for user requests. The input data for this service is provided by PME components across the SOA application.

An end-to-end inter-service policy monitoring and enforcement for SOA. (Chapter 4)
In this chapter, we leverage AOP to design a new framework that is able to monitor and enforce the client's policies at runtime. To achieve this goal, we design PME components that are able to capture the interactions among services and provide input for the TM service. This framework enables us to detect illegal service invocations and malicious services. In the monitoring mode, PME component only monitors the interactions among services passively and reports malicious activities to the user. In the enforcement mode, the PME framework is able to block or redirect the services according to the policies.

An end-to-end intra-service Policy monitoring and enforcement for SOA based on taint analysis. (Chapter 5)
In this chapter, we propose a new fine-grained monitoring framework for SOA, which is able to track the flow of sensitive information inside services. This framework, which is designed based on AOP, prevents the leakage of sensitive data, while it is being transferred between intermediary services. In the inter-service monitoring, we capture *all* interactions among services and every call to untrusted service is considered as malicious. However, we do not know about the flow of information inside the service or whether user's sensitive information is leaked to other services through its interactions. We by introducing intra-service monitoring we leverage a taint analysis mechanism to track the flow of information inside a service and prevent any potential data leakage.

An Adaptive and secure service composition engine (ASSC) for SOA. (Chapter 6)
In this chapter, we provide an application for the proposed architecture, by showing how it can be used to provide adaptation and secure service composition in SOA. We adapt and use an efficient algorithm for knapsack problem to find the most trustworthy service composition among available services in multiple categories while meeting the cost and delay constraints of the overall composition. We demonstrate how ASSC is integrated to the rest of system by rearranging the topology for higher trustworthiness of service composition.

1.5 Assumptions and Scope

The following topics are outside of the scope of this dissertation:

- *Client-side weaknesses and attacks.* This topic is extensively addressed in the literature in the context of web security.
- *Infrastructure (cloud and virtualization) security problems.*
- *Specific orchestration technologies.* In this dissertation, we address generalized scenarios without relying on any specific orchestration technology. This generalization, makes the proposed solutions suitable for microservices, PaaS, and SaaS scenarios too. There are multiple studies in the literature that target the security issues of ESB [2] and BPMN [3].
- *Identity management.* Identity management is an extensive topic that is not in the scope of this dissertation.

1.5.1 Attack Model

Since security is never perfect, it should always be described in terms of threat model and robustness. A threat is some adverse action against the services, data, stakeholders, etc., protected by a security solution. We assume web service developers are not trusted, however, the *service infrastructure providers* (cloud providers or service providers) are not malicious.

Furthermore, the attackers that we consider in this project may have the following capabilities: (1) attackers may gain full control of a certain number of services in a domain, (2) insider attackers in a cloud platform could modify a service, and (3) attackers may have full access to the in-transit messages. We assume that the attacker does not compromise the integrity of the trust manager and the PME components.

The proposed PME component will be deployed outside of the application server (or any other web service containers) at the level of JVM (Java Virtual Machine). Therefore, if attackers obtain access rights at the level of application server configuration, they will not be able to bypass the PME component. On the other hand, if an attacker gets an administrative access to the whole system, then they may be able to bypass the PME component as they can disable any other security measure in that machine.

We further assume either the software stack (OS and JVM) and the PME component are protected by hardware-based *trusted computing* techniques [4] or any tampering with them can be detected by remote attestation techniques [5]. These assumptions are realistic and several successful attempts has been reported in the literature [6–8]. The sketch of these approaches is as follows. They ensure that the server machine boots securely [9] and then a *trusted platform module (TPM)* verifies the integrity of the OS and applications as they are loaded. Any tampering with the OS or JVM can be detected and reported back to a corresponding trust server. Moreover, IBM has attempted to design a virtualized TPM that targets the virtualization and cloud computing. This technology is called vTPM [10], which fits in providing trusted computing for the cloud and SOA. Using the cloud providers' support for trusted computing facilities (e.g. vTPM [10]) and remote attestation of the PME component, we can achieve a reasonable amount of security assurance for these mission critical subsystems. In such cases, when attackers tamper with a

PME component, the trust manager can detect such an attack and starts a recovery process.

Finally, we assume the point-to-point connection between services are secured. This assumption can be realized by using standard transport layer security protocols (such as SSL and TLS) or defining and using an custom digest HTTP header, which holds the cryptographic checksum of the other headers or content. We will further discuss the design of this custom digest header in chapter 3.

1.6 Dissertation Organization

In chapter 2, we explain the proposed security architecture for SOA and we discuss how different system components interact with each other to provide end-to-end security. We discuss the new composite trust models and the trust manager service in chapter 3. In chapter 4, we propose a new policy monitoring and enforcement framework based on AOP to monitor and enforce service consumers' policies. An intra-service monitoring and enforcement based on taint analysis is proposed in chapter 5. Chapter 6 discusses the ASSC as an efficient algorithm for secure service composition problem. We conclude with our conclusions and future work in chapter 7.

2 A POLICY-BASED SECURITY ARCHITECTURE FOR SOA

In this chapter we outline an extensible policy-driven security architecture for end-to-end security in SOA. First, we discuss the limitations of the traditional security solutions in SOA. Second, we propose a security architecture for SOA and discuss the components that are involved. Third, we explain how the system components coordinate with each other to provide an end-to-end security in SOA. Next, we explain the policy subsystem, which makes the proposed architecture highly flexible. Finally, we discuss the alternative architectural design choices and conclude the chapter with related work.

2.1 Design Principles

During this dissertation, we have followed various guidelines in designing and implementing the proposed security measures. The key design principles are as follows:

DP1 *Holistic design.* One of the main limitation of the existing security solutions for web services and SOA is the fact that security is addressed as a set of individualized solutions. For example, existing monitoring systems are separate from other security solutions (e.g., enforcement, trust, etc.). However, to prevent inconsistencies and misconfiguration, security of a system must be addressed by a *holistic* solution.

DP2 *Technology independence.* Many of the existing security tools are tightly dependent of a specific technologies or a vendor. For example, most of the SOA frameworks provide a form of monitoring mechanism. But, besides being limited, they cannot be configured to interoperate with each other. Therefore, the proposed security mechanisms must be technology-agnostic and generic. It means they must be able to work with both SOAP-based and RESTful web services. Moreover, they must be compatible with various frameworks or programming languages (Java, .Net, etc.)

DP3 *Flexibility and adaptability.* Custom security features have been hard-coded into a majority of the traditional applications and services. Therefore, maintaining security across services (or service domains services) is costly if not impossible. This problem is even more challenging in SOA as every service is potentially provided by an independent service provider. The proposed security architecture must be able to adapt to fast-changing requirements of complex SOA systems. Moreover, it must be easy to adapt the system in an structured way (policy-based architecture.)

DP4 *Incrementally deployable and backward compatibility.* This principle is crucial for the success of any practical and real-world solution, since in such scenarios we are facing heterogeneous and legacy systems that are built by various vendors on top of different technologies.

DP5 *Non-intrusiveness and transparency.* In general, security measures require to access and inspect various critical points of a program. Therefore, due to this requirement, security related codes will be scattered over the whole service source codes, which makes them more prune to bugs and less maintainable. The proposed solution must require minimal changes (in source code, runtime environment, etc.) in the existing services.

DP6 *Source code independence.* This is highly desirable if the security framework can function without requiring access to source code (or modify it) as in most of the real world scenarios it is unlikely to obtain access to the source code of web services.

DP7 *High-performance and scalable.* Most of the modern SOA systems are large. Therefore, scalability is an important design principle for these systems.

2.2 Proposed Security Architecture for SOA

In this section we first investigates a set of requirements that our proposed security architecture will satisfy. Second, we discuss the proposed architecture and its components.

To address the challenges discussed in the previous section, the proposed end-to-end security framework must satisfy the following security requirements:

SR1 Defining a new security architecture that is extensible and comprehensive (holistic) to cover various security measures.

SR2 Defining a new dynamic trust scheme for SOA that captures the trustworthiness of services.

SR3 Detecting or preventing disclosure of sensitive information to untrusted services (Confidentiality of data and integrity of the service results).

SR4 Making services accountable for their behaviors.

SR5 Enabling service consumers to define their own security policies.

SR6 Detecting and isolating services, which violates the defined policies and show unexpected behaviors.

SR7 Being responsive to threats and changes in services.

SR8 Having an acceptable level of overhead to be scalable and practical for real-world scenarios

SR9 Providing security-aware service composition (isolating the malicious services).

The proposed security architecture is presented in Figure 2.1. The main components of the system are as follows:

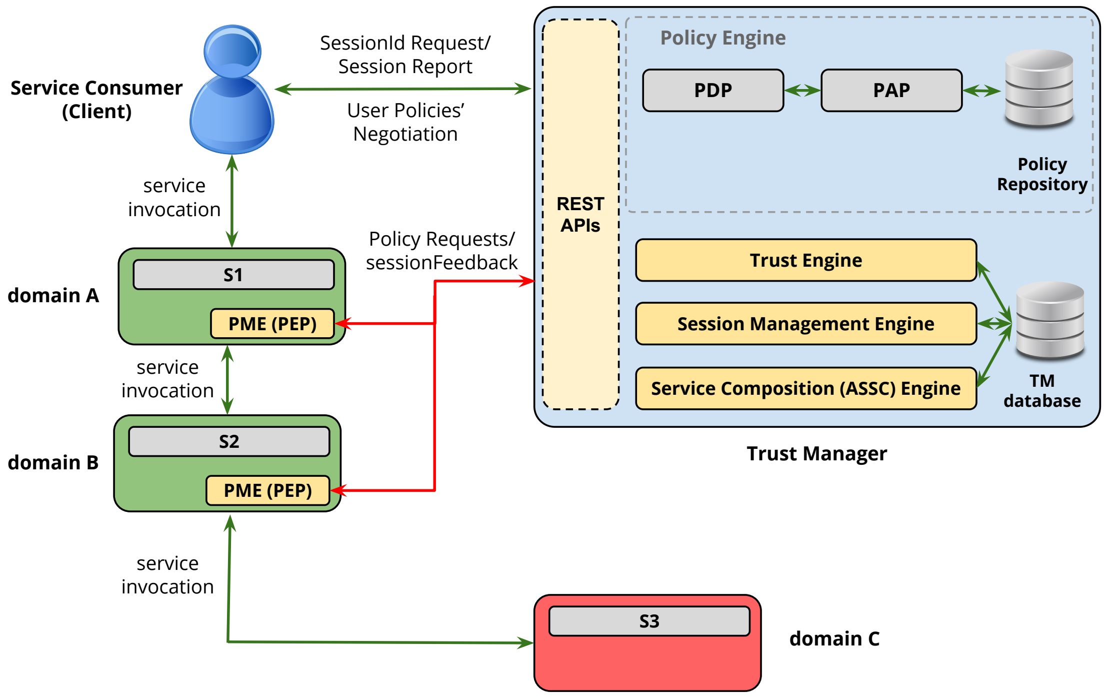


Figure 2.1.: Proposed end-to-end architecture for SOA security

Trust Manager. Trust manager is a neutral trusted third party (TTP) entity that provides the following functionalities:

1. *Trust evaluation and maintenance.* The main responsibility of TM is to calculate and maintain composite trust for a given set of services in SOA. In Chapter 3, we will design and discuss the composite trust algorithms and trust engine component.

2. *Session management.* Session management component is responsible for creating and maintaining a session for every request from service consumer. This session keeps and updates all services that participated in the service invocation. The design and implementation of session management is discussed in chapter 3.

3. *REST API.* TM is implemented as a RESTful web service and all of it's functionalities are accessible through a standard REST interface. This feature makes the TM easily accessible to service consumers, PME components and system administrators.

4. *Supporting XACML policies.* To support generic XACML policies, we leveraged WSO2 Identity Server as an open source policy engine. The main components of XACML policy engine are PAP (policy administration point) and PDP (policy definition point). We will briefly discuss these components in the next sections. TM communicates with this policy engine through web service interfaces.

5. *Adaptive and Secure Service Composition (ASSC).* ASSC component provides a secure and adaptive service composition adaptation in SOA. It leverages an efficient algorithm to find the most trustworthy service composition among available services in multiple categories while meeting the cost and delay constraints of the overall composition.

6. *Storing the service data.* Trust manager database is used to store the trust and feedback information.

7. *Attack detection component.* Similar to ASSC component, several other components could be added to the TM. We will provide a brief outline for an advanced attack detection component in chapter 6. But, the design of this service is not in the scope of this dissertation.

PME component (PEP). PME components are responsible for monitoring the execution of services at runtime to detect malicious service invocations or malicious data leakage (in case of intra-service monitoring and taint analysis). Whenever an external service invocation is going to happen, the PME component sends a feedback message to TM. When PME provides enforcement in addition to monitoring, the PME components stops the service from executing and waits to hear back from TM. The response from TM instructs the PME component to either block the service call or resumes it as normal execution. The PME components could be considered as decentralized policy enforcement points (PEPs) in the vocabulary of the XACML.

2.3 Component Interactions

To simplify and improve the interoperability among various components of the system, presented in Figure 2.1, we designed and used RESTful interfaces. In this section, we discuss the interaction between different components in the proposed architecture.

Figure 2.1 illustrates the operation of the proposed framework for a sample scenario. In this scenario, client calls service S_1, S_1 invokes S_2, and finally, S_2 calls S_3. The service domains that have PME components enabled are shown with green boxes in Figure 2.1. This means that we are able to directly monitor those services for any malicious activity. The interactions among various components of the system are as follows:

1. *Client requests a session identifier (sessionId) from trust manager.* The `sessionId` is a globally unique identifier, which is assigned by TM through a REST API (`/createsessionid/{clientId}`). Once a request succeeds, TM creates a session, which will be responsible to collect the corresponding session feedbacks. SessionId is used by Client and PMEs to maintain session and report feedbacks to TM. The detailed design of the session management will be presented later in this chapter.
2. *Client uploads its policies to policy engine.* Client creates its XACML policies that must be used during monitoring and enforcement and uploads them to the policy engine and then enables them. We will demonstrate several policies in the next chapters.
3. *Client invokes the first service S_1.* In this step, client creates a HTTP request and adds `clientId` and `sessionId` as HTTP headers to the request and then

invokes the service in service domain A. Each service belongs to a service domain , which is controlled by an independent authority. If there are multiple initial services to call, client may ask TM for a service with the highest trust.

4. *S_1 invokes another service S_2 in service domain B.* In service domain B, PME component intercepts the incoming request and extracts the clientId and sessionId for further session feedback. Right before invocation of S_2 by S_1, the PME component intercepts the target URL (`invokedKey`) and sends a feedback request to TM through a REST call. The sessionFeedback is a tuple with four parts: `(sessionId, invokerKey, invokedKey, metadata)`. The `invokerKey` is the current service that calls another service (S_1 in this example). the `metadata` includes contextual information including information leakage (when intra-service monitoring is activated). To improve the performance of PME in monitoring scenarios, we send `sessionFeedback` calls asynchronously in another thread. But, in enforcement scenarios, PME stops the execution of service and waits for the response from TM. If the TM response is PERMIT, then PME resumes the execution of the service. If the response of the TM is BLOCK, then it terminates the current service execution to prevent the malicious service invocation.
5. *S_2 invokes another service S_3 in service domain C.* This step is similar to previous step, except in this case the service will be blocked as we assume S_3 is untrusted. PME in this step also sends another session feedback to TM. In the next chapter, we will discuss how TM will manages and uses these session feedbacks.
6. *Service response to user.* If the invocation is not blocked by PME, the response of the SOA invocation is sent directly to the user.
7. *Client contacts TM for the session report.* After the invocation is finished, client can contact the TM through a REST call to get the session report. Session report includes the composite trust value (overall trust) of the session and

whether any policy has been violated during the service orchestration. The session report also notifies the Client if any information leakage as happened.

2.4 Policy Engine (PE)

Matt Bishop in [11] defines a security policy as *"...a statement of what is, and, what is not allowed."* Schneider presents a more formal definition of a security policy as a boolean predicates on sets of program executions that may disallow their executions [12]. Policy is a mean by which clients specify their security criteria and the way the target services should behave. In most of the enterprise services, security policies are hard-coded into the applications. These ad hoc security policies are hard to modify or maintain. To address this challenge, we leverage a standardized and expressive policy language called XACML into the security framework. The XACML engine decouples the capturing, management, and maintenance of the policies across different components from the rest of the proposed architecture.

Since during this dissertation, we have stressed on reusing the available standards whenever they fit into the design, we carefully examined various policy standards. Finally, we chose XACML 3.0 standard [13], which is a modern, open, and expressive policy language that satisfies all of the desired requirements. There are several choices of open source XACML policy engines. SunXACML [14] was the first implementation of XACML that is used as a basis for more recent XACML implementations. But, unfortunately, it doesn't support the latest specification of XACML. Balana [15] is another open source implementation of XACML, which implements the latest draft of XACML (XACML 3.0). WSO2 Identity Server [16] is another open source project, which is based on Balana and provides more functionalities such as web service API and identity management. We initially implemented a prototype using Balana. But, since WSO2 Identity Server [16] provides web service API, we decided to use WSO2 Identity Server as our policy engine in the experiments.

A Brief Overview of XACML

The eXtensible Access Control Markup Language (XACML) [13] is an expressive and open policy language, which defines a few policy components and the way policies must be defined. Conceptually, XACML is considered as an attribute-based access control (ABAC) paradigm, which is a generalization of role-based access control (RBAC). The main attributes of a XACML policy are: user, resource, action, and environment. A XACML policy is composed of rules, a target and an obligation. The target specifies a set of decision requests that are evaluates by a rule or policy. The effect of a rule is a Permit, Deny, NotApplicable, or Indeterminate.

XACML provides a conceptual policy enforcement architecture (depicted in Figure 2.2) through a set of policy components defined as follows:

- *PEP (Policy Enforcement Point).* The PEP is a component that applies the decision made by policy engine. As a part of this operation, PEP checks the obligations in the XACML responses, which states all the actions that must be taken during the enforcement of policy. In our security framework, PME represents the PEP.
- *PDP (Policy Definition Point).* The PDP, is a place that defines and enables the policies. PDP, evaluates the enabled policies against the incoming requests and returns a XACML response.
- *PAP (Policy Administration Point).* The PAP entity stores and deploys the policies and makes them available to the PDPs.
- *PIP (Policy Information Point).* The PIP, stores the attribute (e.g., information about subjects, roles and groups).

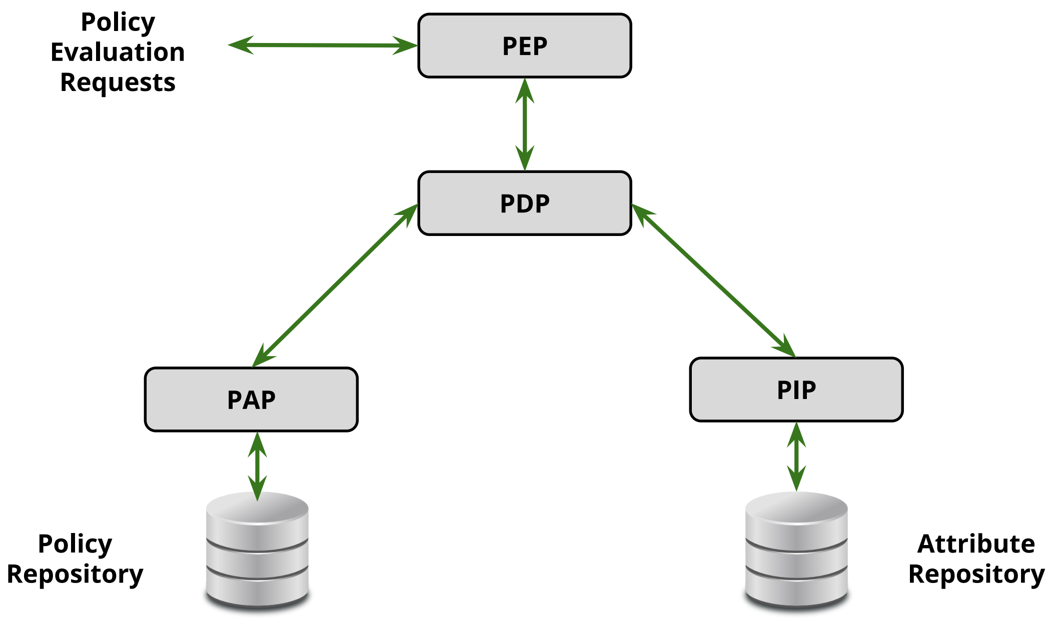


Figure 2.2.: Components of the XACML policy engine

As shown in Figure 2.2, whenever an access request for an object or invocation is intercepted by the TM, it creates a request message and sends it out to the PDP. The XACML request includes all information that is required for the decision making by PDP. Next PDP makes a decision based on a set of predefined XACML policies and sends a XACML response back to the TM. The outcome of the policy evaluation may be permit (resume the execution of service), block (prevent the service from further execution), or redirect (select another service with higher trustworthiness to replace the target service).

Origin of policies. In an enterprise SOA application, there are various stakeholders involved, which each of them may have its own policies. In general there are three main sources of policies:

- *Global policies.* This category includes federal regulations that requires all companies to comply with An example of these policies is HIPPA [17], which will be discussed in chapter 6. Another subset of these policies are administrative policies that are applied to all domains and services. All global policies override other classes of policies.

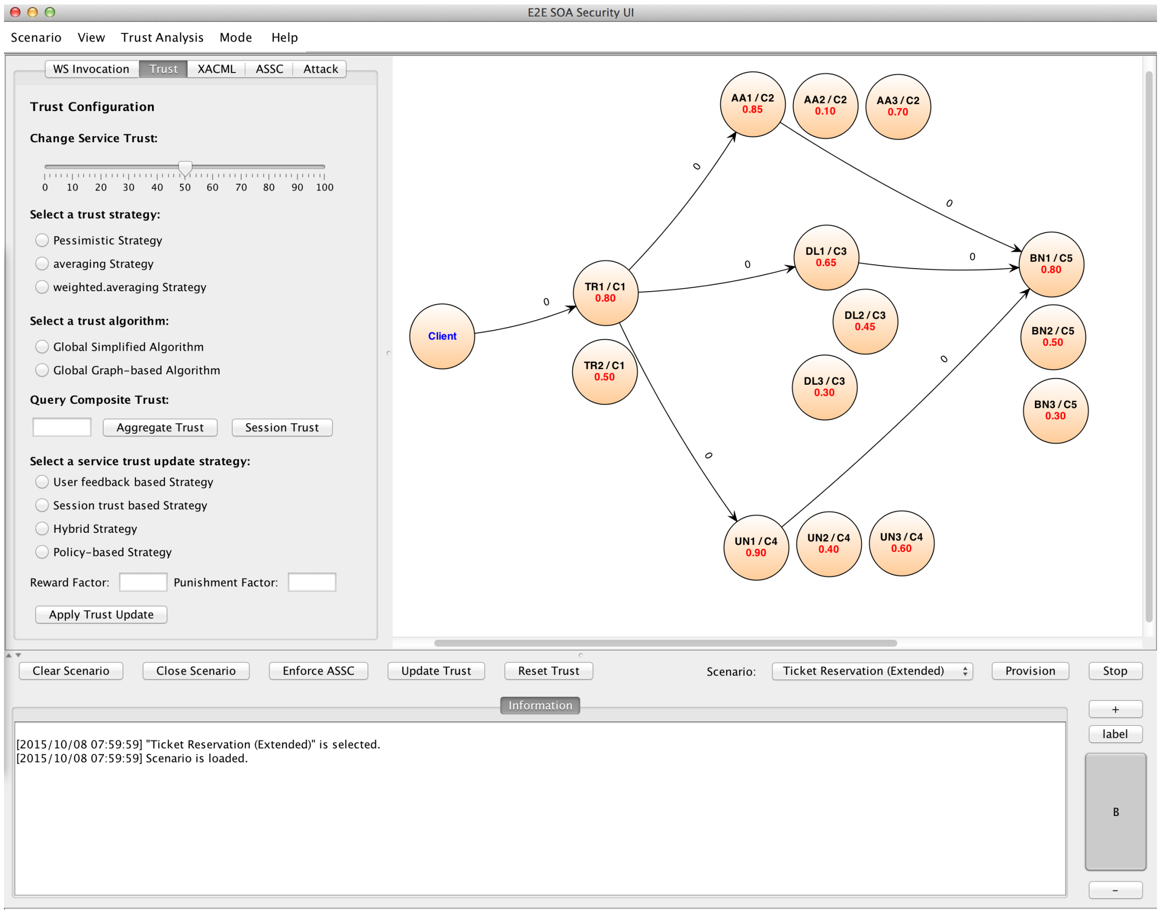
- *Service providers' policies.* These policies are provided by the organization, which is responsible for a service.
- *Service consumers' policies.* These policies are clients' policies, which capture their requirements such as expected trust level of the service.

Multiple sources of policies and each source may have multiple policies. We can take advantage of *policySet* notion to implement multiple set of policies.

We will demonstrate various policies in chapters 4.

2.5 User Interface

We developed a graphical user interface (GUI) to simplify the interaction of clients with the security framework. Furthermore, this GUI enables the users to create, deploy, and study arbitrary SOA scenarios. A screenshot of this GUI is shown in Figure 2.3.



Scenario: Ticket Reservation (Extended)

```
[2015/10/08 07:59:59] "Ticket Reservation (Extended)" is selected.
[2015/10/08 07:59:59] Scenario is loaded.
```

Figure 2.3.: A screenshot of the user interface during the ticket reservation scenario.

The main functionalities provided by the user interface are as follows:

- Interactive creation of new scenarios from arbitrary directed acyclic graphs (DAGs).
- Saving and retrieving created scenarios to and from custom XML files.
- Deploying a scenario by automatically creating and configuring web services.
- Client interface to query the composite SOA scenarios and showing the response.
- Uploading the policy files to policy engine and enabling user to add, enable, disable, and remove policies from PDP and PAP.
- Configuring the composite trust strategies and algorithms
- Enables client to query TM for composite trust of an arbitrary service

- Enables client to update the trust value of services based on selected strategy and given punishment or reward factors.
- Enabling or disabling policy monitoring and enforcement in runtime.
- Enables the QoS configuration and querying ASSC for optimal service composition.
- Clients can interactively change of trust values of every service and enable or disable attacks. This powerful features enable users to study the effect of different trust algorithms and attacks on multiple scenarios.
- enabling clients to rate a session (client feedback) and showing the average rating of a service composition
- showing the session report (notifies client about the potential violations of policies and the trustworthiness of the composite service.)

2.6 Discussion

In this section, we discuss a few topics related to the proposed architecture.

ESB and SOA Orchestration Engines. Since the target of this thesis is to propose a security framework that is generic and applicable to a wide range of real-world services and configurations, we decided to drop the assumption of having enterprise service bus (ESB) in the system. However, as we previously shown in [18], the proposed solutions works for the SOA infrastructures based on ESB too. Similarly, we have no assumption on having BPEL-based or BPMN-based service composition engines. These technologies are not widely used in the industry.

Scalability. In the proposed architecture, trust manager (as a central trust entity) plays a key role in maintaining the end-to-end security. However, it could limit the scalability of the system as it provides multiple functionalities for all services in the SOA application. Addressing the scalability of the TM is not in the scope of this dissertation. However, the following ideas will result in a more scalable architecture:

- Clustering the trust manager. By leveraging the clustering techniques, we can distribute the system load among multiple machines and scale the system. This solution requires synchronization and replication of the policies and trust database. Similarly, we can duplicate the policy engines. For example, [19] discusses a distributed PDP.

- Utilizing the policy caching and push technology [20] for the relevant policy distribution. Caching the policies is highly effective in improving the performance of the system as it cuts two RTTs (round-trip times) and decreases the load of the policy engine. In this solution, trust manager, pushes the *relevant* policies to the participating PME components during the creation of a session. Therefore, service domains can execute the policy engine locally. However, there are two challenges in implementing this model. First, when some policies change in the global PDP, the relevant updated policies must be pushed again to the PME components. Second, policy engine must be protected against tampering similar to the PME components (which will be discussed in the next section).
- Leveraging the cloud computing. Using cloud infrastructures enable the trust manager to scale automatically based on the load of the system.

Relation to cloud computing. The proposed architecture is fully compatible with cloud computing. On one hand, we can leverage the cloud computing to deploy components such as trust manager. In chapter 4–6, we will present a few experiments that demonstrates the deployment of TM in the Amazon EC2 cloud. On the other hand, the end-to-end security architecture can be applied to secure the SOA applications, which are developed based on various cloud delivery and deployment models [21]. We distinguish between cloud service provider (*CSP*) and service provider (*SP*). We define CSP as a infrastructure provider, for example, Amazon is a CSP for EC2 cloud. We also define SP as a software service provider that develops and deploys a platform (e.g., Google for AppEngine) or a software service (e.g., users of Amazon EC2). In some cases, the same company, as in Google AppEngine, may manage both of the cloud infrastructure and software stack.

The main deployment models in cloud computing are public, private, and hybrid. And, there are three cloud delivery models: SaaS (Software as a Service), PaaS

(Platform as a Service), and IaaS (Infrastructure as a Service) [21]. The proposed security architecture can fit into different deployment and delivery models as follows:

1. *Private cloud.* The goal of the security architecture in the private deployment model is to provide internal policy monitoring and detect potential violations due to system inconsistencies or internal attackers. Since the whole deployment is inside a single organizational boundary, therefore, SP and CSP are the same organization and there is no need for a trusted third-party (TTP).
2. *Public cloud.* In the public cloud model, the whole infrastructure in under the control of one or more third-party CSPs. Therefore, to implement the security architecture there is a need for a TTP, which provides the trust manager functionality. Moreover, CSP must cooperate with the TTP to deploy the PME components in a secure way. However, there is no trust assumption on the SP. In fact, the goal is to monitor the SPs.
3. *Hybrid cloud.* This model is the most generic deployment model. In this setting, SOA is distributed among a few public CSPs. The goal of security architecture is to provide an end-to-end policy monitoring (accountability) and enforcement across all these CSPs and their hosted services. Similar to the public cloud, CSPs must cooperate with the TTP.

2.7 Security Analysis of the Architecture

In chapter 1, we presented and discussed the attack model for this dissertation. In this section, we will further focus on analyzing the security of the proposed architecture itself and then discussing how we can improve the security of the presented components.

DoS (Denial of service) attacks against trust manager. The trust manager component is potentially susceptible to DoS (denial of service) or DDoS (distributed DoS) attacks [22]. The goal of these attacks is to make the service unavailable to its authorized users (service consumers and PME components). Fortunately, these attacks have been extensively investigated in the literature and industry. Therefore, we assume the standard DoS countermeasures, such as firewalls and rate-limiting routers, will be used to protect the trust manager from DoS attacks. Moreover, in chapter 6, we present a rate control mechanism using AOP that provides a countermeasure against DoS attacks against the target services.

Tampering or disabling of PME components. We further assume that the service infrastructure providers (could be cloud providers or service providers) are not malicious and they are willing to deploy the PME components and provide a form of *Trusted computing.* The proposed PME component will be deployed outside of the application server (or any other web service containers) at the level of JVM (Java Virtual Machine). Therefore, if attackers obtain access rights at the level of application server configuration, they will not be able to bypass the auditing system. On the other hand, if an attacker gets a root access to the whole system, then they may be able to bypass the auditing as any other security mechanism in that machine. However, using the cloud providers' support for trusted computing facilities (e.g. vTPM [10]) and remote attestation of the PME component, we can achieve a reasonable amount of security assurance for these mission critical subsystems. In such cases, when attackers tamper with a PME component, the trust manager can detect such an attack and starts a recovery process. The trusted computing capability prevents the system from bypassing, compromising, or disabling the PME components. Otherwise, the trust manager can easily detect it through heartbeat mechanism or remote attestation.

We assume either the software stack (OS and JVM) and the PME component are protected by hardware-based *trusted computing* techniques [4] such as TPM (trusted platform component) or any tampering with them can be detected by remote attestation techniques [5]. These assumptions are realistic and several successful attempts has been reported in the literature [6–8, 23].

Compromised or malicious service consumers. In the real world scenarios, service consumers (clients) may act maliciously either intentionally or unintentionally (by being hijacked by attackers). The compromised clients may initiate a DoS attacks on trust manager, which was discussed earlier. Further, they may try to influence the trust mechanism by providing wrong feedbacks. This problem can be addressed by reducing the coefficient for users' ratings or employing a more advanced techniques such as [24].

2.8 Related Work

Web service specifications (or WS-)* [25–28]) are a set of specifications to improve the security of SOAP-based web services. These specifications primarily aim at designing standards for authentication and secure communication in web services. There are many security protocols in this collection that are not fully supported or have not been designed properly. As an example WS-Security, which is the most famous web service security standard, suffers from a severe problem. Even though it protects the security token, however, it does not bind it to the message. It can lead to a class of attacks called XML-Rewriting attacks [29, 30]. In addition to the mentioned limitations, WS-* usually incur an enormous overhead, especially in the chained service scenarios. Moreover, to support these protocols, they need to be tightly coupled into the production code. Finally, these standards are not designed to address the challenges of modern RESTful services or large chained-services. We previously made an effort to use WS-* standards in conjunction with the proposed architecture in [18]. However, since the SOAP web services have lost their popularity to RESTful services, the focus of this dissertation is on the RESTful services.

Authors in [31] present an end-to-end confidentiality based on encrypting custom XML fields using proxy re-encryption. More specifically, they assume every service is associated with a public key and the whole SOA scenario has an external service key associated with it. The trusted third party entity, which is responsible for the key managements, produces re-encryption keys from external public key an all service public keys. In this scheme, all communications are encrypted at the services using the external key and they are routed to the *proxy re-encryption engine* to be re-encrypted. The re-encrypted messages (that are only readable by the target services) are sent to the destination services. There are a number of limitations with this approach. First, since this scheme uses public key infrastructure, it will be very costly to encrypt and decrypt every XML message (or a subset of a message) using public key cryptography. Second, it assumes all communications between services must go through the re-encryption proxy. Since this proxy encrypts all messages, it cannot scale well. Finally, this scheme does not address the problem of information leakage whenever it reaches the destination. For example, if a credit card number of a customer is sent to a destination service, it only protects it from being leaked in the intermediate services. However, the target service, which decrypts the message and extracts the credit card may leaks this information to other untrusted service.

Alternative security architecture would route all the communications among services through a single central service (similar to star topology). In this model, the central service intercepts all messages and enforces all the policies. Authors in [32] attempted to provide an example of this centralized security architecture. However, there are a number of drawbacks in this architecture. First, the central service is a single point of failure, not only for providing the security for the system, but also for all the communications. A large category of services is data-intensive. In this model, the central service must reroute all the communicated data. Second, if the participating services find extra communication channels, the central service will be bypassed easily.

Cisco Enterprise Policy Manager [33] is a framework that tries to provide a XACML-based policy definition and enforcement architecture. However, it has a few limitations. First, this framework is technology-specific and expects all participating business domains to use the Cisco technology (Cisco TrustSec and special switches). Second, this framework is intrusive and all service developers are required to be aware of the Cisco solution and change the code to support it. Therefore, it is not technology-agnostic and cannot be applied to the existing or legacy services. Cisco provides another solution called *Cisco ISE* (Identity Services Engine) [34]. This framework offers a centralized policy control point for comprehensive policy management and enforcement in a single RADIUS-based product from Cisco. However, it suffers from a few shortcomings. First, this technology is designed for a single trust domain. Therefore, it is not capable of maintaining an end-to-end security across security domains. Second, this solution only supports the definition of RBAC and ACL policies that are not as flexible as XACML. And third, it does not support monitoring of interactions among services to provide a dynamic trust framework.

IBM offers a centralized policy management framework, called *Tivoli Security Policy Manager* [35], which is a comprehensive policy management and enforcement framework. Similarly, it supports XACML policy definition and enforcement across multiple business domains. Similar to the Cisco solutions, he Tivoli Security Policy Manager suffers from a few limitations. First, it does not support dynamic trust calculation based on service monitoring. Second, it is not technology agnostic and they expect all participating business domains to use IBM Tivoli technology. And finally, there is no trusted third party in this framework. Therefore, all participating organizations must communicate with each other to provide inter-domain security. In another word, it implicitly assumes a static trust among those service domains.

2.9 Conclusion

SOA paradigm poses new security challenges that cannot be addressed effectively using the existing security solutions that are mainly based on public key infrastructure. To address these challenges, we have designed a new security architecture from the ground up. This architecture, specifies several new components that are necessary to address the end-to-end security challenges in SOA. These components are: PME component, trust engine, session management engine, and ASSC (adaptive and secure service composition) engine. This chapter gives a big picture of the proposed security architecture by first discussing these components individually and next describing their interactions and interfaces. Chapter 4–6 will focus on designing and evaluating these components.

3 TRUST MANAGEMENT

In this chapter, we introduce a new trust service for composite trust evaluation and maintenance in SOA. This framework plays a vital role in providing a holistic SOA monitoring, designing a secure service composition (as we discuss in chapter 6), and enabling system for enforcing policies and defending attacks in runtime.

First, we discuss the motivations behind designing a new trust engine for SOA. Second, we define trust and extract the requirements for a trust engine. In the following section, several trust schemes (single service and composite service trust calculations) are presented based on graph abstraction. We later describe the design and implementation of session management component. Finally, we conclude the chapter by discussing a few potential improvements and related work.

3.1 Motivation

Maintaining trust in SOA provides a safe and low-risk environment for service consumers to dynamically interact and perform their business transactions. Therefore, designing of a relevant and comprehensive trust scheme is vital for the success and adoption of a SOA security framework. SOA promises reusability through chaining services in different ways to perform complex functionalities within or across organizational boundaries. Therefore, secure interactions among services become a challenge. Majority of the current trust management systems are not designed for SOA and usually have wrong assumptions (e.g., transitivity of trust). In traditional web service settings, trust is static and subjective (lack of quantitative or measurable trust), and it is usually provided through public key infrastructure (PKI). Static and qualitative trust schemes are not useful for highly dynamic SOA environments. Additionally, there is no mechanism to handle the trust in chain of services. Another challenge is that service consumers do not have access to a suitable mechanism to select the best service composition among multiple candidate services with similar interfaces offered by different service providers. Finally, services may change their behaviors (internal implementations) without changing their interface, which makes these changes hard to detect by the service consumers. Any of these changes could have a serious impact on trustworthiness of services.

Therefore, modern SOA applications require access to dynamic, quantifiable, and composite trust schemes that take services' reputation and execution history into account. These trust schemes capture all service interactions in SOA scenarios and calculate the trustworthiness of services based on service invocation topologies.

3.2 Trust Definition and Requirements

Trust has been extensively studied in the literature under different contexts [36]. In this section, we first discuss the requirements for designing trust schemes and then we define the concept of trust in SOA.

Requirements for designing trust schemes in SOA. In designing a trust scheme for SOA, we must satisfy the following requirements:

- *Trust must be quantified.* In contrast to traditional trust systems, which are coarse-grained and consider the target as either *trusted* or *untrusted*, we expect trust value gives a measure of a service's overall behavior (not just a predefined subjective measure). There are various sources for quantifying trust that an application may require one or a subset of them. The main sources of trust are: actual execution of a service (execution history) and service reputation (feedbacks given by other service consumers). The trust manager must be able to maintain both trust sources.
- *Trust must be dynamic.* It means even if a service is trusted, whenever it shows malicious behaviors its trust value must decrease. On the other hand, we increase the trust value of a service that is not highly trusted, however, it

consistently shows a trustworthy behavior over a long period of time. Another source of trust is customer rating, which dynamically change the rating of a service. Finally, trust update mechanism may be based on a policy defined by a client.

- *Trust schemes must be pluggable and user-centric.* Different application contexts may have different definition of trust. Therefore, trust management must enable pluggable trust schemes to capture various trust semantics and logics. In such a trust management system, client are able to select a trust scheme or a hybrid combination of them based on the requirements of the target application.
- *Trust scheme must capture the multi-hop nature of the SOA and avoid transitive trust.* Most of the existing web service security models only consider the client-server model for accessing to web services [37–39]. Therefore, they implicitly assume that trust is transitive. Transitive trust means, if service S_1 considers service S_2 as trustworthy and S_2 believes S_3 is trustworthy, then S_1 should trust S_3, which is not necessarily true in multi-domain SOA applications. Therefore, it is necessary to design new composite trust schemes to support the end-to-end security in SOA. To address this requirement, trust manager requires composite trust schemes.

To address these requirements, we first present a definition for trust in SOA, and then in the next section we propose composite trust schemes. The proposed trust schemes are implemented in the trust engine component of the trust manager.

Definition of trust. Traditionally, trust has been defined qualitatively as subjective opinions of clients about the trustworthiness or target services. However, qualitative and subjective definition of trust is not useful for SOA applications. We define trust as a measure that quantifies the likelihood of a service to perform as expected. This expected behavior is defined through public interfaces and SLAs of a service. We define trust as a real value in the range of $[0, 1]$, which 0 represent *not trustworthy* and 1 represents *completely trustworthy.* This definition is closely related to the definition of risk. We can define risk as "$1 - trust$". Therefore, as trustworthiness of a service increases, the risk of data disclosure or illegal service invocation decreases. The given definition of trust could easily be extended to take other parameters into account. For example, the number or strength of security mechanisms that a service support could be a part of a trust value.

Due to complexity of SOA, a single trust value cannot represent the different aspects of trustworthiness. For example, in Figure 3.1, client C has invoked S_1 for 100 times. S_1 has invoked S_2 for 99 times and S_3 once. Individual trust values of services are not good representative for the whole service composition. To overcome this challenge, we maintain three types of trust values:

- *Service trust.* This trust value indicates the inherent trustworthiness of an individual service regardless of other services in the service chain. The service trust changes based on service's execution history, client's rating of the service composition, or session trust. Client will define the trust update mechanism.
- *Session trust.* Session trust shows the *actual* trustworthiness of a single session. This metric is primarily used for reporting the trustworthiness of a specific service-chain invocation to the client. For example, if in the last session of Figure 3.1 service composition, S_1 has called S_2, then the session trust is calculated based on the trust value of S_1 and S_2 only.
- *Composite (or aggregate) trust.* This is a global trust value, which represents (predicts) the *expected* overall trustworthiness of a service composition. For example, another client might want to know the expected trustworthiness of a service composition by calling S_1. In this case, we must take execution history of all participating services (edge weights) into account.

Session trust and composite trust depend on both individual service trust values and the structure of the service composition. We will give concrete examples for the calculation of composite trust in the next sections.

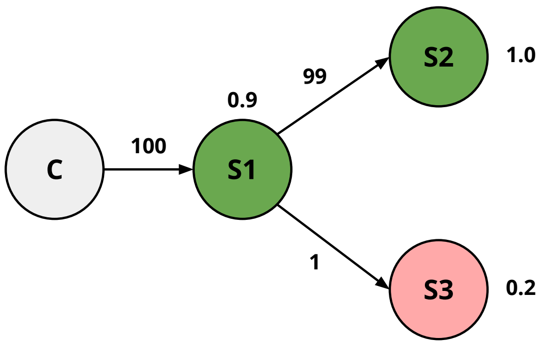


Figure 3.1.: Example scenario for composite trust.

3.3 Graph-based Composite Trust Schemes

previous trust schemes (similar to [18]) calculates the trust values of each service separately. This definition is useful in the context of small and simple topology with at most two hops, where one service consumer interacts with a single service provider. However, such definition is not suitable for a generalized SOA architecture. In SOA, every service transaction is composed of a chain of services (or more precisely, a directed acyclic graph of services), which could include an arbitrarily number of services. Therefore, a more accurate notion of trust for SOA must capture both of the following factors:

1. *Trustworthiness of all services.* It means the trust value of a SOA application is dependent on the trustworthiness of its services.
2. *Structural dependencies among those services.* It means the definition of trust in SOA must capture the way services are interacting with each other.

In this section, we proposed three broad trust strategies (pessimistic, averaging, and weighted averaging), and two algorithms (coarse-grained global algorithm, and graph-reduction algorithm) in order to address the challenge of calculating composite trust. These strategies and algorithms are used to calculate the composite trust based on service graphs. We coin this approach *graph-based composite trust.* Graph-based composite trust schemes do not require any changes in the trustworthiness of individual services (service trust values). As we previously mentioned, service trust changes only based on service's execution history, client's rating of the service composition, or session trust.

Service consumer is given a choice to choose among the proposed strategies and algorithms based on the sensitivity and context of the target SOA application. For example, pessimistic strategy with graph-reduction algorithm produces the best result for the mission-critical scenarios. However, in general web applications, weighted averaging strategy with graph-reduction algorithm would be the most suitable.

In the following subsections, a graph abstraction is presented to capture the complex and arbitrary interactions among services in SOA. This model will be used to represent, calculate, and maintain the trust in SOA.

Definition of service graph (SG). A *service graph* is defined as a weighted directed acyclic graph (DAG), G, with an ordered pair $G = (V, A)$ of:

- V: which is a *set* of nodes. Each node represents a service in SOA. We associate a trust value to each node (service) in this graph.
- A: which is a *set of ordered pairs* of vertices, called *arcs.*[1] Each arc represents an invocation between two services and has a weight, which shows the number of invocations. For example, if S_i has invoked S_j 60 times in the past, the assigned weight to (S_i, S_j) is 60.

Generation and maintenance of service graphs. Trust manager is responsible for creating and maintaining this graph, which is global and captures *all* service interactions in a SOA ecosystem. For each SOA scenario, we assume there is a bootstrap configuration file, which contains a list of certified services with optionally their initial trust value. However, at the beginning there is no arc to connect these nodes. As PME components begin to send invocation metadata, trust manager adds corresponding arcs and maintains the trust value according the chosen trust calculation strategy.

[1]In undirected graphs, these pairs of nodes are unordered and are called *edges.* Throughout this dissertation, we may use them interchangeably.

Definition of service composition graph (SCG). As mentioned, the definition of service graph is global and includes all services in a SOA system. We define *service composition graph* as a local *subgraph* of the service graph. Whenever we query the trust manager for a composite trust of a service S_i, trust manager, creates a sub-graph of the graph graph, which includes S_i and all other reachable services from this service.

3.3.1 Basic Topologies

Any arbitrary service graph is composed of a number of basic topologies. The basic topologies are used in the calculation of the composite trust based on a given trust strategy. The basic service interactions in SOA have the following forms:

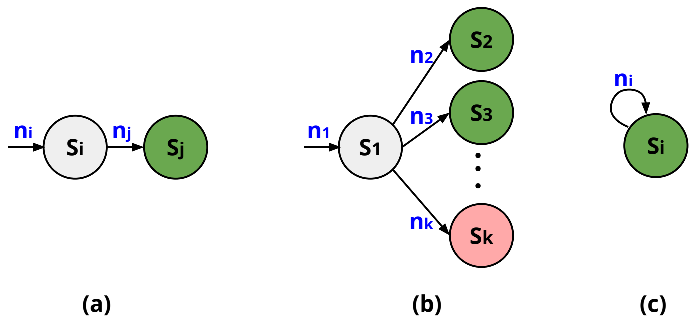


Figure 3.2.: Basic topologies in SOA service graphs: (a) chained subcontract, (b) conditional subcontract, (c) recursive invocation

1. *Chained subcontract.* In this topology, a service is only dependent on a single other service. In Figure 3.2 (a), S_i only requires to call S_j to perform its functionality. The weights on the arcs show that S_i is called n_i times by its predecessors (n_i is the aggregation) and calls S_j for n_j times during the lifetime of service orchestration.
2. *Conditional subcontract.* In the second basic topological form, a service is conditionally depend on a number of other services. In this case, an invocation of

dependent services does not happen every time. Therefore, each arcs shows how many times the target service is being invoked. For example, in Figure 3.2 (b), S_1 is called n_1 times in total. During these n_1 calls, it has called each service S_i for n_i times ($i = 2, 3, ..., k$).

Note. The service graph is generated and maintained entirely based on empirical feedback data collected from PME components and we do not have any assumption on the internal logic of participating services. Therefore, we do not assume $\sum_{i=2}^{n} n_i = n_1$ as every time S_1 is called, it may call any subset of S_2, S_3, ..., S_k arbitrarily.

3. *Self-invocation recursive call.* This topology represents a service that recursively calls itself, which in graph theory they are called *self-loops.* This topology is shows in Figure 3.2 (c).

   To simplify the modeling, we assume recursive calls are part of the service's logic. This simplifying assumption is valid since in this special case, no external service invocation happens, and the trust value of a service that invokes itself remains the same. Therefore, service graphs are *simple directed graphs.* The theoretical foundation for this simplification is based on the fact that any recursive program could be transformed into a semantically-equivalent iterative program [40]. Also, if any service provides multiple functionalities, which is against the design principles of SOA), we will represent that service with multiple logical nodes. This abstraction will prevent the creation of loops.

In the future, we reference the basic topologies as *basic topology (a), (b), or (c).*

3.3.2 Composite Trust Strategies

We have identified three main *strategies* that can be used to compute the composite service trust. No single strategy works best for all possible SOA scenarios. The choice must be made based on application's domain and sensitivity. These trust calculation strategies are as follows:

1. *Pessimistic.* The idea behind this strategy is the *principle of weakest link* [41]. This principle states that the security of a system depends on the security of the weakest component of that system. Based on this principle, the security (trust value) of a chain of services is equal to the security of the least trustworthy service in that chain. Therefore, for any composite service composed of S_2, S_3, ..., S_n services with an arbitrary topology, the composite trust would be calculated as follows:

   $$t_{composition} = \min_{i=1..n} t_{s_i}$$

   We believe that this principle can be applied to critical and high-assurance scenarios, where security plays a crucial role in them (for example, in military scenarios). However, it could be too restrictive for some commercial or less sensitive scenarios. For example, in service mashups [42], a web service may just invoke another service to get the latest weather service and display it on its page. In this scenario, weather service does not play a critical role in the overall functionality of the main service. If the weather service is untrusted, using the pessimistic strategy makes it a dominant service in calculation of the composite trust, which is not reasonable.

   Note 1. It is worth noting that this strategy is not limited to calculating trust values and it could be applied to other metrics too. For example, *throughput of a link* is a metric that naturally requires pessimistic strategy.

   Note 2. Similar to the pessimistic strategy, we could define an optimistic strategy, which find the maximum trust value of the constituent services as a trust value of the composite service. However, since there is no useful use case for this strategy, we omit it.

2. *Averaging.* In this strategy, we define the composite trust as an average of the sub-services' trust values. Given a set of services $S_1, S_2, ..., S_n$, we define the composite trust as follows: $t_{composite} = \sum_{i=1}^{n}(t_{S_i})/n$.

3. *Weighted Averaging.* The intuition behind the weighted averaging is the fact that different services do not have the same effect and importance. We consider two cases, which require assigning higher weights to a subset of services. In the first case, if a service S_i is invoked by more services than service S_j, then it should have a higher weight. The reasoning behind it is the fact that S_i is a more popular service and it should have a higher effect on the composite trust. Similarly, in the second case, if a service S_i invokes a larger number of services than S_j, then it should have a higher weight, as the response of all invoked services will be returned to this service and it has a higher control over a larger subset of service graph. Higher weight of a trusted service will make the service composition trustworthier and vice versa. In fact, we are looking for a weighting mechanism, which amplifies the trust value of a service based on its weight.

   To formalize this factor, we leverage the *PageRank algorithm* [43], which gives a higher weight to nodes with larger *indegrees*.[2] Applying the PageRank algorithm directly to the service graph does not reflect the second case mentioned earlier. To consider the effect of services with large fanout (services that call a large number of other services), we augment the service graph by adding a reverse arcs for every arc in the service graph. Augmenting the service graph reflects the fact that whenever a service S_i calls service S_j, S_j will reply back to S_i, which will be represented as a reverse arc. The output of applying the PageRank algorithm on an augmented service graph (for $S_1, S_2, ..., S_n$) is a list of scalar values $r_1, r_2, ..., r_n$ that $\sum_{i=1}^{n} r_i = 100$. The r_i values represent the weight of each service. We slightly modify these values by replacing the $r_i < 1$ with 1 (for simplification and practical purposes).

   Now that we selected proper weight coefficients, we provide the formula for calculating the composite trust using weighted averaging strategy. For a service

[2] *Indegree* of a node is the number of incoming arcs of that node. In the context of service graph, it shows how frequently a service is invoked by other services.

graph composed of n services, $S_1, S_2, ..., S_n$, the composite trust is calculated as follows: $t_{composite} = \sum_{i=1}^{n} r_i.t_{S_i}$.

3.3.3 Coarse-Grained Global Algorithm

In this algorithm we do not take the complicated structure of the service graph into account and we assume all sub-services have the same importance in the overall trust. Even though this assumption reduces the accuracy of the final estimated composite trust, however, it allows a faster execution and it is suitable for very large scenarios. This algorithm first creates a subgraph of SG by extracting all services reachable from S_{entry} in the service graph. S_{entry} is the first service that is initially called by service consumer. The subgraph creation can be implemented by using BFS (Breadth-first search) [44].[3]

Algorithm 1 Coarse-grained global algorithm to calculate the composite trust.

```
 1: Input:
 2: SG: The service graph $G = (V, A)$
 3: $S_{entry}$: An entry service from which the composition trust is calculated.
 4: strategy: Composite trust strategy.
 5: Output:
 6: $t_{composite}$: a composite trust of a subgraph with root $S_{entry}$.
 7: Step 1:        ▷ Extracting the service composition graph (SCG) from SG.
 8: SCG ← subgraph(SG, $S_{entry}$);        ▷ SCG is a subgraph of SG reachable from
    $S_{entry}$
 9: Step 2:        ▷ Calculating the composite trust based on the predefined strategy.
10: if strategy = pessimistic
11:     $t_{composite} \leftarrow \min_{i=1..n'} t_{s'_i}$
12: else if strategy = averaging
13:     $t_{composite} \leftarrow \sum_{i=1}^{n}(t_{S_i})/n$
14: else if strategy = weighted_averaging
15:     $t_{composite} \leftarrow \sum_{i=1}^{n} r_i.t_{S_i}$
16: return $t_{composite}$
```

[3]We can replace the BFS with DFS (Depth-first search) or any other reachability algorithms.

To compute the complexity of this algorithm, we have to compute the complexities of its components. There are three components involved in this algorithm. The first component is BFS algorithm with $O(|V| + |A|)$ complexity. The second component is the complexity of applying composite trust strategies. Given the required weight coefficients, all three strategies are linear $O(n^{'})$ $(n^{'} \leq n)$ that $n^{'}$ is the number of services in the subgraph (SCG). The final component of the complexity is calculating and processing of the PrageRank weights. There are various algorithms with different complexities in the literature. However, the complexity none of them is higher than $O(|V| + |A|)$.[4] Therefore, the complexity of the coarse-grained global algorithm is $O(|V| + |A|)$.

3.3.4 Graph-Reduction Global algorithm

The coarse-grained global algorithm does not capture the structure of the service graph and it simply applies an operation on a list of services. Although this algorithm is useful for simple topologies and fast composite trust estimation, it is not suitable for the precise calculation of composite trust in generic topologies. As illustrated in Figure 3.1, the graph structure and services' invocation probabilities affect the calculation of a composite trust.

To overcome the limitations of the coarse-grained global algorithm, we propose a new algorithm, which is based on graph abstraction. The idea behind graph-reduction global algorithm is the fact that every arbitrarily complex graph is composed of a set of basic topologies (Figure 3.2. Therefore, by replacing a basic topology in a graph by a vertex (service) with an equivalent trust value and weight, we can iteratively reduce a service composition graph to a single vertex with a trust value of the composite trust of the original graph.

In this algorithm, the pessimistic strategy operates similar to the coarse-grained global algorithm. However, the averaging strategy is applied to the basic topologies,

[4]In fact, there are advanced algorithms that calculate the PageRank more efficiently, for example, authors in [45] presented a sub-linear algorithm for PageRank.

Figure 3.2 (a) and (b), as follows:

Topology (a): For basic topology (a), the composite trust is calculated as follows: $t_{composite} = \frac{n_i.t_{S_i}+n_j.t_{S_j}}{n_i+n_j}$. Similarly, the composite trust with weighted averaging strategy employs the PageRank weights as follows: $t_{composite} = \frac{n_i.r_i.t_{S_i}+n_j.r_j.t_{S_j}}{n_i.r_i+n_j.r_j}$.

Topology (b): For basic topology (b), the weight of each service is used as the weight of the respected trust value. The trust value of the total topology is: $t_{composite} = \sum_{i=1}^{k}(n_i.t_{S_i})/\sum_{i=1}^{k} n_i$. The weighted strategy on topology (b) is calculated as follows: $t_{composite} = \sum_{i=1}^{k}(n_i.r_i.t_{S_i})/(\sum_{i=1}^{k} r_i.n_i)$.

One of the main challenges in the graph-reduction global algorithm is to find and calculate the basic topologies in a correct order. The intuition behind the proposed solution is to find a service (or services) in the composite graph that has no outgoing arc (which hereafter we call it a terminating service). By backtracking from a terminating service, we can identify a basic topology. This intuition can be implemented using topological sorting algorithm [44]. Since the service composition graphs are DAGs (e.e. directed acyclic graphs), applying the topological sorting algorithm will *always* produce a list of sorted services. After applying the algorithm to the identified basic topology, we remove all participating services in that basic graph from the sorted list and append a new service with an equivalent weight and trust of the respective composite service.

There is a non-trivial case (illustrated in Figure 3.3 (a)) that cannot be reduced to the basic topologies. This problem happens when the terminating service has $indegree > 1$. To resolve this issue, we duplicate the terminating service (with the same trust value and weight) and create $indegree - 1$ new cloned services. Each of these services will be connected to the predecessor services through a single arc. The initial service in the sorted list will be replaced by new cloned services. After this processing, we can reduce the service graph using the basic topologies. This process must be repeated, if the same problem happens again before termination of the algorithm. Figure 3.3 (b) illustrates this operation. In this Figure, S_4 and S_5

have *indegree* > 1 and should be split. After applying the algorithm, S_4 and S_5 will be replaced by S_4', S_4'', S_5', and S_5''. The operation is presented in Algorithm 2.

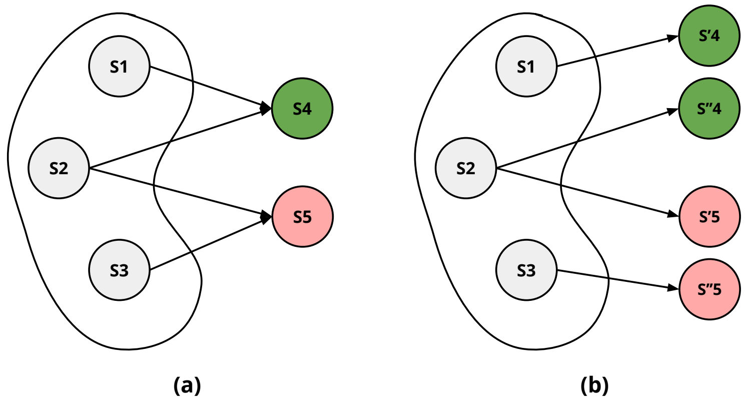


Figure 3.3.: The problem of terminating services with *indegree* > 1: (a) problematic scenario, (b) a solution by cloning the terminating services.

Similar to algorithm 1, the complexity of algorithm 2 is dominated by BFS and topological sorting, which both of them have the complexity of $O(V + A)$. Therefore, the complexity of this algorithm is $O(V + A)$.

3.4 Implementation of Trust Components

Trust manager (TM) is composed of multiple components, as illustrated in Figure 2.1. In this section, we discuss the implementation of three trust-related components: trust engine, session management engine, and REST APIs. TM is implemented as a RESTful web service using Java RESTful API (JAX-RS) and deployed over Jersey framework. All interactions with TM happen through standard REST APIs. The interactions of other components of the system with TM are described in chapter 2.

The raw data that TM operates on is received from PME components across all services in the SOA application. Whenever a service invokes an external service, the respective PME component generates a *PME Feedback*. PME feedback is a message that captures the following metadata: the session ID, invoker service ID, invoked

Algorithm 2 The graph-reduction global algorithm.

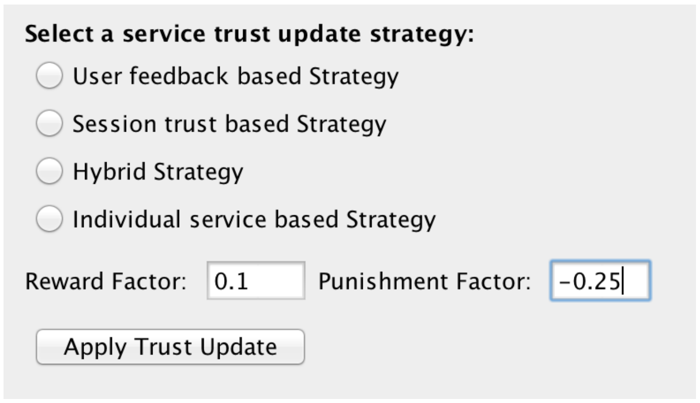
```
1: Input:
2: SG: The service graph $G = (V, A)$
3: $S_{entry}$: An entry service from which the composition trust is calculated.
4: Output:
5: $t_{composite}$: a graph-reduction based composite trust of a subgraph with root $S_{entry}$.
6: Step 1:        ▷ Extracting the service composition graph (SCG) from SG.
7:     SCG ← subGraph(SG, $S_{entry}$);        ▷ $SCG = [S_1, S_2, ..., S_{n'}]$
8: Step 2:        ▷ Generating a list of topologically-sorted services
9:     L ← topologySort(SCG);        ▷ $L = [S'_1, S'_2, ..., S'_{n'}]$
10: Step 3:        ▷ Calculating the composite trust based on the predefined strategy.
11: while $L \neq \emptyset$
12:     $TS = L$.getLast()        ▷ $TS$ stands for terminating service
13:     if $TS.indegree > 1$
14:         ▷ This function applies the solution discussed in Figure 3.3 (b)
15:         $L$ ← resolveByCloning(L);        ▷ Updates the $SCG$; returns a new $L$
16:     ▷ $TOPO = \{S''_1, S''_2, ..., S''_{n''}\}$ is the basic topology that encloses $TS$
17:     if strategy = $averaging$
18:         if (TOPO is basic topology (a))        ▷ In this case $TOPO = \{S''_1, S''_2\}$
19:             $t_{composite} = (n_1.t_{S''_1} + n_2.t_{S''_2})/(n_1 + n_2)$
20:         else
21:             $t_{composite} = (n_1.r_1.t_{S''_1} + n_2.r_2.t_{S''_2})/(n_1.r_1 + n_2.r_2)$
22:     else if strategy = $weighted\_averaging$
23:         if (TOPO is basic topology (a))
24:             $t_{composite} = \sum_{i=1}^{n''}(n_i.t_{S''_i})/\sum_{i=1}^{n''} n_i$
25:         else
26:             $t_{composite} = \sum_{i=1}^{n''}(n_i.r_i.t_{S''_i})/(\sum_{i=1}^{n''} r_i.n_i)$
27:     else if strategy = $pessimistic$
28:         $t_{composite} \leftarrow \min_{i=1..n''} t_{s''_i}$
29:     ▷ Replacing the $TOPO$ with a service, which has a trust value of $t_{composite}$.
30:     replaceTopology($TOPO$);
31:     $L \leftarrow L - TS$;
32: end-while
33: return $t_{composite}$
```

service ID, and optional meta-data (for example, timestamp of this feedback, which could be used in the attack detection sub-system). The PME components send these events to the trust manager. Trust engine is responsible for reconstructing the service graph from interactions between services reported by PMEs. We maintain two type of graphs. First, we maintain a global graph, which ingests all interactions across all sessions. Second, for every session (created by a client for a single SOA service invocation), we create a new session graph, which is solely responsible for that sessions. The goal of global graph (service graph) is to estimate the trustworthiness based on history of services' executions. On the other hand, session graph gives an actual trustworthiness of a specific session. For example, Figure **??** shows the global graph. However, a single session may be a chain of $C \rightarrow S_5 \rightarrow S_7$. The session report is used for reporting to the client.

3.4.1 Trust Engine

Trust engine is responsible for trust calculation and update in SOA. In the previous section Trust calculation algorithms are discussed in section 3.3. We implemented these composite trust algorithms for all trust strategies (pessimistic, averaging, and weighted averaging).

The second part of trust engine is trust update mechanism. trust update is responsible for changing the trust values of services dynamically, based on their execution history. TM could be configured to apply the trust updates automatically or updates the trust value through REST API.


Select a service trust update strategy:

- ◯ User feedback based Strategy
- ◯ Session trust based Strategy
- ◯ Hybrid Strategy
- ◯ Individual service based Strategy

Reward Factor: 0.1 Punishment Factor: −0.25

Figure 3.4.: A screenshot of trust update strategies.

We have implemented four trust update strategies as follows (as illustrated in Figure 3.4):

- *User feedback based strategy.* This strategy uses the user's feedback (or rating) to update the trust value of all services involved in a session. This rating is subjective (based on user's perception of trustworthiness of the outcome of a service call) and is captured as an integer value between 1–5. If the rating is low (i.e., 1 or 2), then all services would be punished based on a specified punishment factor. Otherwise, all services would be rewarded based on a defined reward factor. For example, in Figure 3.4, if user's rating is two, then all services participating in the corresponding session would be punished by reduction of 0.25 from their trust value. If the trust value become negative, then it will be recorded as 0 (similarly, if it becomes greater than 1, it will be recorded 1).
- *Session trust based strategy.* In this strategy, we use the actual composite trust value of a session, which is calculated based on one of the composite trust algorithms, to decide whether contributing services should be rewarded or punished. The decision is based on whether the trust value is less than a predefined threshold (punishment) or greater than threshold (reward). For example, if the composite trust value of session is 0.4 and the trust threshold is 0.5, then, services would be punished by reduction in their trust value.

- *Hybrid strategy.* This strategy, combines both of the previous strategies. If user's rating is low, and the composite trust of a session is less than threshold, then services would be punished. Similarly, if user's trust is high (i.e., 3-5) and composite trust value is greater than the trust threshold, then services would be rewarded. In other two cases, services neither rewarded no punished.
- *individual trust based strategy.* In this strategy, a service is rewarded or punished based on the fact that whether it calls another trusted or untrusted service. If a service calls another service with a trust value less than a predefined threshold, then it would be punished by a defined punishment factor. The difference between this strategy and the second strategy is that in this strategy, we only penalize services that call untrusted services (instead of punishing or rewarding all services in the session).

3.4.2 Session Management Engine

Session Management Engine plays a key role in connecting different pieces of the proposed framework together. Sessions create an end-to-end state, which ties the chain of service invocations to each other and enables the proposed SOA security architecture (presented in chapter 2) to achieve an end-to-end view of a chain of service invocations. Session management has three goals: session creation, session maintenance, and session usages.

Session creation. The first responsibility of the session management engine is to create sessions based on requests from client. `sessionId` is a globally unique identifier, which is created through a REST API (`GET /createsessionid/{clientId}`). SessionIds is used by clients and PMEs to maintain sessions. Once a sessionId is created, trust manager creates a session object and associates it with the sessionId. Every session object is responsible for storing all data related to a session. One session is allocated as a global session, which stores all interactions among all services during the lifetime of a SOA application.

Session maintenance. Session maintenance is composed of two steps: extending a session and updating it. As SOA service invocation progresses (either client calls the first service or a service S_i calls service S_j), the session must be extended to the new invoked service. One of the main design principles in the dissertation is to make no changes in the source code of services. This restriction makes the implementation of service maintenance a challenging task. To overcome this challenge, we proposed a solution based on defining and using a custom HTTP header. Whenever a client calls the first service in the chain, it inserts a custom HTTP header `("sessionId":SessionValue)`, which SessionValue is a new sessionId received from TM.

On the service side, we have implemented the `RequestFilter` class that preprocesses the incoming requests before sending them to the target service. The `RequestFilter` class implements the `ContainerRequestFilter` interface from Jersey framework [46]. Therefore, whenever the service request reaches the first service domain, the `RequestFilter` extracts the sessionId. This sessionId is captured by `SessionIdAspect` class of the PME deployed in this service domain. Similarly, we have defined another aspect called `InvokerKeyAspect` in PME, which captures the IP address of the service exposed by `containerRequest.getBaseUri()` method in the `RequestFilter`. In chapter 4, we will discuss how AOP and aspects work. Another part of session maintenance is to propagate the sessionId to other services whenever the current service calls other services. Similar to the `RequestFilter` mechanism, we have designed `ResponseFilter`, which implements the `ContainerResponseFilter` of the Jersey [46]. `ResponseFilter` enables us to add the sessionId to the HTTP headers of the outgoing requests without changing the source code of the services. The same process with be repeated for the other services in the service chain.

Session Usages. Previously, we discussed how sessions are created and maintained. these sessions have two use cases. The first use case is through PME components. Whenever a service received and processes the incoming request, PME captures the `InvokedKey`, which is the URL of the target service required for the session feedback, and send out a `SessionFeedback(sessionId`, `invokerKey`, `invokedKey`, `metadata)` message to TM in parallel. At the point of sending the feedback, the other parameters are already captured by other aspects as discussed before.

The second use case is generating a session report for the client. After the invocation chain is finished, client can contact the TM through a REST API to get the session report. Session report includes the composite trust value (overall trust) of the session and whether any policy has been violated during the service orchestration. The session report also notifies the client if any information leakage as happened.

3.4.3 Trust Manager REST API

In this section, we preset an overview of the REST APIs provided by the trust manager:

- `GET /createsessionid/{clientId}.` This request is used by clients to initiate a new session. Client, should include its own identifier to the request and will receive a unique session ID.
- `POST /sessionfeedback.` This request is used by PME components to send a feedback tuple to TM. The format of the tuple is: `(sessionId, invokerKey, invokedKey, metadata).`.
- `POST /userfeedback/{sessionId}/{rating}.` This request it is used by users to set their rating for the current session.
- `POST /trustupdate/{sessionId}/{rewardPunishFactor}.` This request is used by clients or TM itself to reward or punish services in the current service composition.
- `GET /policyrequest/{resource}/{subject}/{action}/{encodedMetadata}.` This request is used by PME components to query for authorization of a service request in enforcement scenarios. Resource is the target service, which is going to be called. Subject is the current service, which is going to invoke the resource. Action is invocation of the service. EncodedMetadata, is a encoded

value of metadata, which shows occurrence of information leakage. The TM, once it receives this request, queries the policy engine and returns the response to the originator PME.

- `GET /getcompositetrust/{sessionId}/{serviceName}/{trustAlg}/{trustStrategy}`. This request is used by clients to get the composite trust value of a service. Trust engine requires trustAlg and trustStrategy parameters to calculate the composite trust. The detailed explanation will be presented in chapter 4.
- `GET /getassc/`. This request is used by clients to get an optimal service composition for the current set of services. The details will be provided in chapter 7.
- `GET /isleaked/{sessionId}`. This request is used by client to check whether any information leakage has happened in a given session.
- `GET /isviolated/{sessionId}`. This request is used by client to check whether any violation of policy has happened in the given sessionId.
- `GET /getaverageuserfeedback` This request is used by client to get the average rating of the service composition given by all users in all sessions.
- `GET /sessionreport/{sessionId}`. This request is used by client to get a summary of events happened in the given session. This summary includes: potential policy violations, information leakage, and the actions (including blocking and redirecting) taken to prevent them.
- `POST /setservicetrusts` This request is used by client or system administrator to set the initial trust values of services.
- `POST /setconstraints` and `/setresources`. These requests are used by client to set QoS and cost parameters required by ASSC for optimal service composition selection.
- `GET /getservicetrusts`. This request is used by client/UI to receive the latest service trust values (not composite trust values).
- `POST /setpolicytrustconfig`. This request is used by client to configures the policy request at the TM side.

- `GET /getserviceweights.` This request is used by client/UI to get the number of invocations happened between every pair of services. This will update the edge weights on the UI.
- `GET /removesession/{sessionId}.` This request is used by client to remove a given session from TM.
- `GET /resettm.` This request is used by client to reset the TM.

3.5 Related Work

We originally proposed and implemented a single-service trust scheme in [18]. In this model, trust value of every service is calculated only based on its actual past execution history and its reputation. This scheme calculates the trust values of each service isolated from other services. For example, in Figure 3.1, S_1's trust is calculated only based on its decision to invoke one of the immediate S_2 or S_3. It means, trust value of S_1 does not reflect what might happen down in the service invocation chain. Trust evaluation component adapts *exponentially weighted moving average (EWMA)* as a trust update mechanism [47].

Authors in [48, 49] study the dynamic trust evaluation in SOA. Malik [48] introduces a framework called RATEWeb for trust-based service selection and composition based on peer feedback. This framework proposes a distributed and reputation-based trust algorithm. The shortcoming of this approach is that they do not address the multi-hop composite trust and their notion of trust is only based on client's feedbacks without taking the actual execution of services into account. Spanoudakis et al. [49] propose an approach to keep track of trusted services by collecting data about their compliance with the SLAs. The collected data is composed of both service events and clients' feedbacks. However, their solution depends on a new event-driven architecture which changes requires major changes in the services and their deployment environments. Moreover, approaches similar to [48, 49] are not suitable for SOAs with a lot of services because the monitoring system would need to collect intensive information from a lot of peers and clients, which would make it very expensive.

We propose a lightweight and flexible trust mechanism based on feedbacks from the auditing system to maintain the trust levels of different services.

In addition to the runtime behavior monitoring of services, we also constitutes the user's feedback (rating or recommendation) in our trust model. We call this feedback the *reputation-based component* of trust, which demonstrates the user's perception of the trustworthiness of the invoked service. In our model, we assume that a user will rate the invoked service after completion, by choosing a number from $[0, 1]$. The reputation of a service could be maintained by averaging over all reported users' ratings. It also could be used in the calculation of overall trust of a service. There are more advanced reputation-based trust schemes reported in the literature (e.g., [50, 51]), which are outside the scope of this dissertation.

3.6 Conclusion

Trust management is a key component in designing an end-to-end security framework for SOA and it lays a foundation for the operation and interaction of other security components in the system. Every trust management system is based on a conceptual trust model, which specifies how trust should be defined, maintained, and leveraged. In this chapter, we discussed that the current trust models (based on public key infrastructure) are not suitable for SOA. To address this problem, we have designed and implemented a dynamic and flexible composite trust model based on graph abstraction, which captures the complex interactions among all services and maintains three different types of trust metrics in SOA. These metrics are service trust, session trust, and composite (aggregate) trust. These separate trust metrics serve different purposes. For example, individual services are used by the ASSC (presented in chapter 6) to calculate a service composition with the highest level of trustworthiness; Session trust is reported to the client as an indicator of the trustworthiness of the current session; The composite trust is used by the policy engine and trust manager to detect and prevent data leakage attacks.

The proposed trust model is flexible and it provides several trust strategies and trust update mechanisms. For example, this model supports three trust strategies (pessimistic, averaging, and weighted averaging), which enables the clients to tune the trust model based on the application domain. For example, in weighted averaging, we use the PageRank algorithm to calculate the weight of each service based on their connectivity importance. To further increase the flexibility of the proposed trust model, we support multiple trust update mechanisms. The first one is reputation-based, which updates the trust values based on the ratings from clients. The second update mechanism is based on the composite trust of the corresponding session. This mechanism penalizes or rewards all participating services in a session based on the composite trust of the session. The third update mechanism is hybrid, which combines the previous mechanisms. The last update mechanism updates the trustworthiness of a service based on its actual behavior (whether it calls untrusted services or not). Clients may choose any combination of these trust strategies and trust update mechanisms based on their application domain.

Another contribution of the work in this chapter is the design and implementation of an end-to-end session management engine. This component enables the creation and maintenance of service sessions non-intrusively (without requiring any changes in the participating services). To achieve this property, it leverages custom HTTP headers and intercepts the REST requests through AOP, before they enter their target service.

We have implemented the proposed trust model as a RESTful web service called trust manager. Trust manager *automatically* creates and maintains service graphs from incoming session feedbacks, which are generated from runtime monitoring of services in an SOA application by PME components. The flexibility of the proposed trust model makes it suitable for a wide range of SOA applications (from mission-critical military scenarios to non-critical commercial applications).

4 INTER-SERVICE POLICY MONITORING AND ENFORCEMENT

In this chapter, we propose a policy monitoring and enforcement (PME) framework for SOA, which is able to inspect the execution of services and report the predefined events to trust manager. This framework also enables the SOA clients to enforce their policies at runtime. The structure of the chapter is as follows. First, we investigate the motivations and requirements for designing a policy-based monitoring and enforcement framework for SOA. Second, we provide a brief background on aspect-oriented programming (AOP) and discuss how it enables us to implement the PME framework for SOA. Next, we discuss the enforceable policies and the enforcement strategies. Fourth, we describe the process of designing a wide range of policies using the AOP-based framework. Next, the effectiveness and practicality of the proposed framework is demonstrated through a ticket reservation case study. Finally we conclude the chapter by related work.

4.1 Motivation and Requirements

SOA paradigm encourages designers to develop applications composed of services from various service providers. Despite all advantages of SOA, there are a number of challenges that must be addressed. The main challenge is that SOA applications span across organizational boundaries, which makes them susceptible to violation of service consumers' policies (such as disclosure of private data to untrusted services). Policy violations might happen due to financial or malicious incentives or it simply might happen unintentionally (because of software bugs or misconfiguration). Furthermore, there is no end-to-end service accountability in current SOA systems. For example, a service may act maliciously by leaking personally-identifiable information (PII) to untrusted services without being detected for a long time. Unfortunately, none of the widespread security protocols (such as SSL and TLS) are able to guarantee the safety of their data. The reason is that protocols like SSL provide point-to-point communication security, and once data enters a service, they will no longer protect it from malicious behaviors. In fact, the agility in the development of modern SOA applications and web services, which leads to short release cycles and constant changes, makes the current cryptographic and static solutions less effective.

On the other hand, service consumers are getting increasingly more concerned about the safety and privacy of their data and they expect service providers to comply with their policies. However, without an independent *verification mechanism*, there is no way to measure to what extent a service provider delivers its promises. More specifically, an inter-service policy monitoring and enforcement framework for SOA must meet the following requirements:

- *Platform independence.* It must be technology-agnostic (independent of the underlying web service technologies) and platform-independent.
- *Transparency.* It must be as transparent as possible to the service providers and users. Any intrusive mechanism that requires access to source code of services or changes in them, may not be possible (or it may be too restrictive and incur an excessive amount of maintenance cost.) Therefore, this requirement is essential for adoption in industry.
- *Flexibility.* It must be easy to extend and define new policies and reconfigure the framework to use them, without requiring static service certification.
- *Expressiveness of policies.* It must be able to define and monitor a large category of policies from different sources (service provider, service consumers, and federal organizations).
- *Non-bypassability.* since AOP framework works on top of JVM and is not directly accessible by the programs or services, therefore, they cannot bypass the proposed PME components unless they have administrative access to the system, which in that case no security solutions cannot withstand the potential tempering.

- *High performance.* AOP framework is very high performance compared to Bytecode or instruction-level solutions. The reason is that the rewriting of a class file is performed during the load-time of a class (when a class is loaded for the first time). After the initial rewriting, every interception causes a very small processing overhead. We further evaluate the effect of AOP solution in section 4.5.2.

To address these requirements, we proposed a policy monitoring and enforcement framework, which is able to inspect the execution of services at runtime to detect, prevent, and report the policy violations. This framework has two parts:

- An inter-service monitoring and enforcement (also called coarse-grained or service-level information flow control). This framework is able to inspect the interactions *among* services and *detect* malicious activities. Additionally, it is able to *prevent* malicious activities before happening. To achieve this goal, we take advantage of the TM service (as discussed in chapter 3) as a trusted third party, a policy engine, and PME components to implement a *reference monitor* for SOA.
- An intra-service monitoring and enforcement (based on taint analysis). This framework focuses on tracking the flow of private data from *within* a target service and it is able to verify whether this service leaks this sensitive information and prevent it. This framework be the topic of the next chapter (chapter 5).

4.2 Background on Aspect-Oriented Programming (AOP)

Aspect-oriented programming (AOP) was originally proposed to complement Object-oriented paradigm by separating and encapsulating crosscutting concerns from the main logic of the applications [52]. AOP enables us to inspect and modify the functionality of a program at compile time or runtime. More specifically, we can add features, orthogonal to the current logic of a program. In this context, orthogonality means we do not need to modify the existing source code of a program. Here we present a brief description of major AOP concepts:

Joinpoint. A joinpoint is a specific event (or point) in the control flow of a program. There are different types of joinpoints including method, field, and constructor, joinpoints. For example, in Java, events such as calling of a method, accessing or modifying a field, and execution of a constructor are joinpoints.

Pointcut. A collection of joinpoints is called a pointcut. Pointcuts are AOP expressions (similar to regular expressions) that match particular joinpoints. More specifically, they define at what points during the execution of a program, we are interested to be notified to inject our custom logic (which is called an *advice*) to the control flow of that program.

Advice. Advice (similar to an event handler) is a method that is executed in response to a pointcut match event.

Aspect. An aspect is a reusable component that encapsulates crosscutting concerns and it is a bundle of one or more pointcuts and advices. *Logging* is not tied to the logic of programs and crosscuts almost all components in a program. Therefore it is an example of a crosscutting concern that can be implemented as an aspect.

Weaving. The process of injecting the aspect code into the main functionality of a program is called weaving.

Invocation. An invocation is a class that represents a joinpoint and its relevant context (talk information about the called method, its arguments, etc.) at runtime.

AOP frameworks provide a pointcut language to define pointcut expressions. They specify in which joinpoints an advice must be invoked. AOP joinpoints are highly expressive. They can virtually intercept any potential event in the execution of a service. Here we demonstrate the expressiveness of the joinpoints through a few examples in JBoss AOP [53]. The description of the Figure 4.1 is as follows:

```
 1 edu.purdue.cs.soa.C1
 2 edu.purdue.cs.soa.*
 3 $instanceof{edu.purdue.cs.soa.I1}
 4
 5 public * edu.purdue.cs.soa.C2->m1(java.lang.String, *)
 6 public !static void edu.purdue.cs.soa.C3->*(..)
 7 void $instanceof{edu.purdue.cs.I2}->m2(T1)
 8 T2 edu.purdue.cs..->m3()
 9 * edu.purdue..->*(..)
10
11 public edu.purdue.cs.soa.C4->new(java.lang.String)
12 !public edu.purdue.cs..->new(..)
13
14 public edu.purdue.cs.soa.C5->f4
15 !public edu.purdue.cs.soa.C6->*
```

Figure 4.1.: Examples of pointcut expressions

- *line 1.* This joinpoint matches the class `C1` in package `edu.purdue.cs.soa`. Lines 1–3 are called type patterns.
- *line 2.* This pattern matches every class in package `edu.purdue.cs.soa`.
- *line 3.* This joinpoint matches every class that is instantiated from interface `I1` in package `edu.purdue.cs.soa`.
- *line 5.* This joinpoint matches method named `m1` from class `C2`, which is in package `edu.purdue.cs.soa`. This method has two parameters. The type of the first parameter must be `String`, but the second parameter may have any type. Finally, this method must be public and may return a value with any type. Lines 5–9 are called method patterns
- *line 6.* This joinpoint can be interpreted similar to the previous pattern. But, it matches any method (with any number and type of parameters) within class `C3`. The matching method cannot return a value or it cannot be static.
- *line 7.* This pattern matches method `m2` from any class that is instantiated from `I2` interface.

- *line 8.* This joinpoint matches method named `m3` from any class within the package `edu.purdue.cs`, but not the sub-packages. For example `edu.purdue.C5.m3` will be matched.
- *line 9.* This pattern is highly generic and matches any method with any return value, and any type and number of parameters from classes inside the package `edu.purdue`.
- *line 11.* This joinpoint matches creation of an object from class `C4` through a constructor wich accepts a `String` argument. Lines 11–12 are called constructor patterns
- *line 12.* This pattern matches creation of an object from any class within package `edu.purdue.cs` using a non-public constructor.
- *line 14.* This joinpoint matches access to a public field named `f4` from objects of class `C5`. Lines 14–15 are called field patterns
- *line 15.* Finally, this pattern matches access to any non-public fields from objects of class `C6`.

All of the pointcut expressions provided in Figure 4.1 could be combined using logical operations (e.g., and, or, not, etc.).

Pointcut definition. Pointcuts can be defined by wrapping a pointcut expression in one of the following specifiers:

- *execution(method or constructor).* This pointcut, weave the methods at their entry-point. Whenever those points are reached, the AOP framework invokes an advice. For example, `execution(public !static void edu.purdue.cs.soa.C3->*(..))` intercepts the execution of a pointcut expression presented in line 6 of Figure 4.1.
- *construction(constructor).* Similarly, this pointcut wraps an object creation expression. For example, `construction(!public edu.purdue.cs..->new(..))` wraps line 12 of Figure 4.1.

- *get or set (field expression).* These two pointcuts intercept the access (read and write) to the fields. For example, `get(public edu.purdue.cs.soa.C5->f4)` wraps line 14 expression of Figure 4.1.
- *call(method or constructor).* The call pointcut has a slight difference from execution pointcut. The call pointcut intercepts a joinpoint by weaving the caller site (a place where the target method is invoked), while the execution pointcut weaves the called site (right at the beginning of the invoked method). An example of a call pointcut is presented in line 8 of Figure 4.1.
- *withincode(method or constructor).* The withincode pointcut matches any possible joinpoints inside a defined method or constructor. For example, `withincode (T2 edu.purdue.cs..->m3())` matches all the joinpoints inside the method specified in line 8 of the Figure 4.1.
- *within(type expression).* The within pointcut, matches all joinpoints inside a specified type (class or interface). For example, `within(edu.purdue.cs .soa.*)` matches any joinpoints inside any class that is defined in package `edu.purdue.cs.soa` as specified in line 2 of Figure 4.1.
- *all(type expression).* This pointcut matches all constructors, methods, and fields inside a specified type. For example, `all(edu.purdue.cs.soa.C1)` matches all types of joinpoints inside class `C1`.

Main types of advices. AOP enables the monitoring system to intercept a set of predefined and custom events in a service and trigger a suitable functionality based on those events. There are three main types of advices in AOP:

1. *Before.* Whenever a joinpint is matched, the AOP framework calls this advice right before proceeding with that joinpoint.
2. *After.* advices of the type after are executed after the joinpoint returns normally.
3. *Around.* Around advice combines both before and after advices. In fact, it wraps the target joinpoint and has control on whether to execute the matched joinpoint or not.

```
<bind pointcut="execution(* *->(..))">
   <before aspect="MyAspect" name="myBeforeAdvice"/>
   <around aspect="MyAspect" name="myAdvice"/>
   <after aspect="MyAspect" name="myAfterAdvice"/>
   <throwing aspect="MyAspect" name="myThrowingAdvice"/>
   <finally aspect="MyAspect" name="myFinallyAdvice"/>
</bind>
```

Figure 4.2.: Five types of advices and their binding to a pointcut in JBoss AOP

The Figure 4.2 shows how to can bind one or more advices to an specific pointcut. Specifically, it asks the AOP framework to execute the advices, which are methods of an aspect class (`MyAspect.myBeforeAdvice(..)`, etc.), at all possible pointcuts of the project. However, this generic pointcut is not recommended in real world scenario as it incurs a huge overhead to the program. The reason is for every line of the program we may need to run one of more advices. The general recommendation is to limit the scope of the pointcuts as much as possible to reduce the overhead of the AOP.

Stateful aspects. In an aspect, when a pointcut is matched, in response one or more advices are executed. Usually, these advices work based on the locally available metadata. However, if we extend the aspects to keep a state, we are design and support more interesting policies by observing the execution of a service during a time window (and not only a single instance of execution). The aspect state can be maintained at the process-level (by using static classes or fields), or at the interprocess-level (by persisting the state in a file or a database). In the next chapter, we design an intra-service monitoring framework, which extensively uses these stateful aspects.

4.3 Designing Monitoring and Enforcement Mechanisms

In this section, we first compare the static and dynamic monitoring approaches and justify the selection of dynamic (runtime) mechanism for SOA security. Second, we briefly explain the paradigm and concepts of aspect-oriented programming (AOP).

Finally, we justify the usage of AOP for SOA monitoring and present a systematic approach to achieve this goal.

4.3.1 Choosing AOP as an Implementation Tool

Monitoring frameworks are designed based on one of the following approaches:

1. *Static analysis.* In this category, the service's source code or binary is analyzed to find the potential security risks and policy violations. The goal is to ensure that all services behave *as specified*. They generally use formal modeling techniques to prove a service or program provide certain properties (such as [54–56]). There are several drawbacks to this approach. First, this mechanism has limitation in addressing the real world services as they are too complex to be modeled precisely in formal languages. Second, this approach usually requires a service to be rewritten. This modified program has some checks to be performed at runtime. This approach violates a few of the requirements discussed in section 4.1, including transparency, platform independence. Additionally, approaches based on static analysis cannot monitor or enforce most of the real world policies as they are not decidable without runtime execution. Furthermore, dynamic trust calculation is a key component for end-to-end SOA security, which cannot be maintained using static analysis.
2. *Dynamic and runtime analysis.* In this category, identify some critical points during the execution of a service, which there is a chance of potential policy violation. The next step during the runtime of a service is to call a procedure on those events to inspect the method and the context of the execution to verify nothing malicious happens. For example, whenever a dangerous method (for example a method, which does network operations) is called, it must be inspected at runtime to make sure no private information is leaked to untrusted services.

To satisfies the requirements listed in section 4.1, we selected the second approach and chose AOP as a tool to implement it. The design goals and capabilities of the AOP are fully aligned with our requirements for a holistic monitoring and enforcement mechanisms in SOA. In this section, we discuss why and how AOP can be used to design a monitoring framework.

In general, security monitoring features are *crosscutting concerns*. This means adding a security mechanism to a project is independent of the application logic. For example, authenticating and authorizing users of a service is independent of its functionalities. At the same time, AOP promotes the idea of decoupling the crosscutting concerns to make the large-scale systems more modular and maintainable. AOP realizes this goal by pushing the crosscutting concerns into *aspects* and precisely defining the points where the relevant *advices* must be applied. Therefore, we take advantage of the AOP to implement the end-to-end security architecture in SOA independent of the participating services. This feature is highly valuable in practice as modern SOA systems are distributed and complex. Otherwise, we would require to implement the security features scattered over various component of the system, which would make the whole system more error-prune and harder to maintain.

The second advantage of using AOP in implementing security solutions is the narrow *semantic gap* between security domain and AOP concepts. For example, some security solutions try to monitor an application by analyzing the low-level Java Bytecode using Apache BCEL [57] and Javasnoop [58]. Compared to these solutions, AOP provides higher-level constructs (pointcut abstractions) to inspect the services and inject new security checks at runtime.

An AOP-based framework satisfies most of the requirements listed in section 4.1 as follows:

- *Platform independence.* AOP is a highly generic concept and there are AOP frameworks designed for almost all well-known programming languages and platforms [59]. For example, authors in [60] designed an AOP framework for GCC compiler. AOP-based aspects are generic and they could be easily adapted to work with new technologies or execution frameworks. We demonstrated this framework for both RESTful and SOAP-based [18] technologies under different web service containers (such as JBoss Application server and Jersey).

- *Transparency.* AOP based solution does not require access to source code and has a minimal impact on the existing SOA applications. This property makes it possible to easily apply the proposed security solutions to legacy applications.
- *Flexibility.* AOP enables us to extend the system easily by designing new aspects to support new policies or new features. Since the framework is non-intrusive, such extension could be added in a modular and systematic way (pluggable). Moreover, we can add or remove new policies at runtime without interrupting the operation of the services.
- *Non-bypassability.* since AOP framework works on top of JVM and is not directly accessible by the programs or services, therefore, they cannot bypass the proposed PME components unless they have administrative access to the system, which in that case no security solutions cannot withstand the potential tempering.
- *High performance.* AOP framework is very high performance compared to Bytecode- or instruction-level solutions. The reason is that the rewriting of a class file is performed during the load-time of a class (when a class is loaded for the first time). After the initial rewriting, every interception causes a very small processing overhead. We further evaluate the effect of AOP solution in section 4.5.2.

4.3.2 AOP-based Monitoring and Enforcement Mechanisms

The proposed PME framework is implemented and as a number of *PME* (policy monitoring and enforcement) components and their interactions with the TM service. The PME components are deployed in target service domains to monitor the execution of services. These components are extensible and pluggable and allow the addition or removal of new *security aspects.* Security aspects enable a PME framework to support new policies flexibly. The design process of these security aspects is composed of four stages. These stages are as follows:

1. *Identifying triggers.* In this step, we identify a list of potentially dangerous classes and their relevant methods that could be misused to perform malicious activities in the target service. The outcome is called an *inspection set* or *triggers.* In Java, the triggers could be selected from standard JDK APIs or third-party libraries' APIs. Potential triggers could be networking, RMI, JMS, and file access APIs. Depending on the application infrastructures, we may need to select a list from third party communication libraries (for example, Netty and JBoss remoting). Interestingly, it is possible to automate the identification of trigger classes. A potential solution would first traverse the package structures and class files in the service jar files, and then analyze the class-files using `javap` [61] disassembler to find a subset of trigger classes that are available in a specific service. Another approach would be to inspect the running services through tools developed by Java Virtual Machine Tool Interface (JVM TI).
2. *Designing pointcuts.* We design a set of pointcuts to match the specified triggers in the previous step. We will take advantage of the AOP's expressive pointcut definition capability.
3. *Extracting contextual metadata.* At this stage, we define one or more advices that analyze the *context* of the invocation and decide whether the current invocation is malicious or not. The context could be any information at the joinpoint that could help with deciding the maliciousness of the operation. For example, when an external network operation is intercepted, we require to extract the invoked URL as a contextual information. The contextual metadata is mainly taken from the arguments of the invoked method (method pointcuts) or the state that we maintain in the aspect. The corresponding advice may make some decisions locally or may require to create a standard XACML request (which encapsulates the collected contextual metadata) and send it to the policy engine to check the conformity of the operation to the specified policies.

In our implementation, we decided to push the policy engine interface to the TM service. The PME components capture the context and contact the TM service through the *sessionFeedback* REST API. In the monitoring scenarios, the TM only records the incident and service without any further action. In the enforcement scenarios, the PME component first sends a session feedback and then calls the *getPolicy* REST API to receive the policy response. In the meantime, the execution of service is paused. The TM service contacts the policy engine, which evaluates the request based on active policies and notifies the PME component about the outcome of the evaluation. To improve the performance, this step can be designed asynchronously (using multi-threading or non-blocking APIs) in the monitoring scenarios.

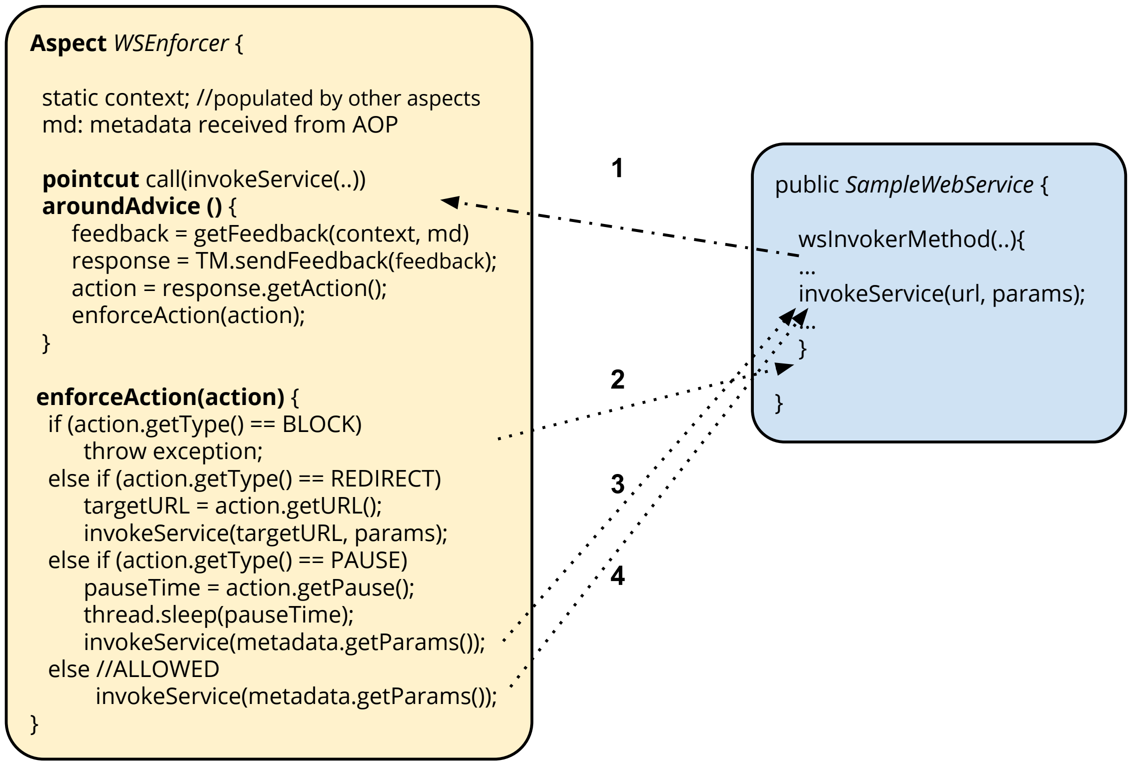
4. *Taking proper actions.* In enforcement scenarios, if the response from TM service or policy engine indicates that some policies are violated, then the specified action must be taken (such as stopping the service). The details of the enforcement strategies are discussed in subsection .

In Figure 4.3, we illustrate how we can use AOP to intercept all service invocations within a service. This figure shows that the `SampleWebService` invokes another web service through `wsInvokerMethod()`. This operation is performed by calling a library method called `invokeService()`. To intercept this service invocation scenario, we design an aspect called `WSMonitorAspect`. Every aspect has a pointcut definition and one or more advices. We define a pointcut to precisely capture the invocations of `invokeService()`. The next step is to define proper advices to encapsulate the logic of what we want to do once this pointcut is matched. We define an `around` advice which encapsulates the target service invocation. The order of events is as follows:

1. Before the invocation of `invokeService()` method, the corresponding pointcut (`call(invokeService (..))`) is matched and therefore, `aroundAdvice()` will be called.
2. The `aroundAdvice()` encapsulates what we want to inspect before calling the `invokeService` method. In this example, we extract the context information

and then call the TM service. TM on behalf of the PME component contacts the policy engine (through a XACML request with relevant metadata) and returns the response. If the operation is not allowed, then . At the end of this step, the control flow returns to the `invokeService` method to be executed. At this point, the pointcut gets matched again and the `aroundAdvice()` is called. If the report flag is enabled, it means that the service has violated the policies and it must be reported back to the trust manager (with relevant metadata for further analyses).

3. In the last step, the control flow of the execution returns back to the service to continue its execution.


```
Aspect WSEnforcer {

  static context; //populated by other aspects
  md: metadata received from AOP

  pointcut call(invokeService(..))
  aroundAdvice () {
     feedback = getFeedback(context, md)
     response = TM.sendFeedback(feedback);
     action = response.getAction();
     enforceAction(action);
  }

 enforceAction(action) {
  if (action.getType() == BLOCK)
       throw exception;
  else if (action.getType() == REDIRECT)
       targetURL = action.getURL();
       invokeService(targetURL, params);
  else if (action.getType() == PAUSE)
       pauseTime = action.getPause();
       thread.sleep(pauseTime);
       invokeService(metadata.getParams());
  else //ALLOWED
        invokeService(metadata.getParams());
}
```

```
public SampleWebService {

  wsInvokerMethod(..){
  ...
  invokeService(url, params);
  ...
  }

}
```

Figure 4.3.: Enforcement of policies for service invocations using AOP

4.3.3 Enforcement Strategies

In this section we investigates the potential actions that a PME component is capable of performing in case of a policy violation. The main *enforcement strategies* are as follows:

- *Aborting the service.* In this strategy, the PME component prevents the service from continuing its execution by raising an unchecked exception or calling the `System.exit()` method. This strategy is interruptive and terminates the whole SOA application. In this strategy, the untrusted service would be added to a blacklist to prevent future selection by other clients.
- *Replacing or redirecting the service.* In this strategy, instead of terminating the service and the whole SOA application, the PME component replaces the untrusted service with a trusted service in the same category during the runtime. This mechanism enables the security framework to isolate the malicious services by redirecting all their incoming requests.
- *Throttling the service (rate control).* This strategy will only pauses a service invocation for a limited time. This may be helpful to control the traffic to a target service. Another use case would be to slow down the invocations when the trust manager observes a malicious pattern. This delay may last until this unusual activities is analyzed.
- *Rearranging the topology dynamically.* This strategy, may replace multiple services at once to maintain a required property. For example, if a highly trusted service gets compromised, the security framework may decide to replace two services with the lowest trustworthiness by two other services to maximize the trustworthiness of the overall service composition. The algorithms discussed in chapter 7 may be used to find the optimal composite service at any time and use the list of candidate services with this strategy.
- *Retrying service invocation.* In case of high system load and intermittent service outage, this strategy may retry to invoke the next service for a predefined number of times.

- *Executing the service normally.* If no policy is violated, then this strategy will be applied.

4.4 Designing Policies for PME Framework

In 2.4 we discussed the system architecture and briefly mentioned the policy engine. The TM service creates a XACML request and sends it to the policy engine. The policy engine evaluates the request and responds back to the TM. The TM service will parse the XACML response and returns a REST response to the PME component. In this section, we further discuss this component and explain the policies we designed for the ticket reservation service case study.

In this dissertation, we have use WSO2 Identity Server [16] as an open source policy engine. WSO2 Identity Server implements the XACML 3.0 specification [13], which supports policy obligations and advices. *Obligations* are a set of one or more actions that must be executed by PME components (as PEPs) in response to an access request. For example, the decision for an access request may be *permit*, however, PDP may ask the PME component to log the operation and send it to the trust manager for future analyses. This request for logging is called obligation. *Advice* is similar to obligation except for the fact that it is not mandatory to be performed by the PME component.

Figure 4.4 shows a sample XACML policy with an obligation. Lines 2–10 define the target of this policy. A policy target specifies under what condition it must be evaluated for a XACML request. In this example, policy target matches any request with a resource value that matches the specified regular expression `localhost:90[0-9][0-9]`. This regular expression matches all local addresses with port value of 9000–9099. In general, a policy might have an arbitrary number of rules, but in this example, we only define one rule. This rule defines a new attribute `http://cs.purdue.edu/soa/trust` for the environment category (lines 15–19). If a request that matches the target and has this attribute with value less than 0.5, then this rule is evaluated as *Deny*. Otherwise, the result will be *Not Applicable* or *Indeterminate*. If this rule evaluates as *Deny*, then PDP insert an obligation with value *block* in the policy response.

```
<Policy xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17" PolicyId="
    policy_trust" RuleCombiningAlgId="urn:oasis:names:tc:xacml:3.0:rule-combining-
    algorithm:deny-overrides" Version="1.0">
  <Target><AnyOf><AllOf>
    <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-regexp-match">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
    localhost:90[0-9][0-9]</AttributeValue>
      <AttributeDesignator
          AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
          Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
          DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true"
    />
    </Match>
  </AllOf></AnyOf></Target>

  <Rule Effect="Deny" RuleId="service-invocation">
    <Condition>
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:double-less-than">
       <AttributeDesignator
          AttributeId="http://cs.purdue.edu/soa/trust"
          Category="urn:oasis:names:tc:xacml:3.0:attribute-category:environment"
          DataType="http://www.w3.org/2001/XMLSchema#double"
          MustBePresent="true" />
       <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#double">0.5</
    AttributeValue>
      </Apply>
    </Condition>
    <ObligationExpressions>
      <ObligationExpression FulfillOn="Deny" ObligationId="obligation_block">
        <AttributeAssignmentExpression AttributeId="obligation_attr">
             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
    block</AttributeValue>
        </AttributeAssignmentExpression>
      </ObligationExpression>
    </ObligationExpressions>
  </Rule>
</Policy>
```

Figure 4.4.: Example of a XACML policy with obligation

Table 4.2 presents nine policies that are developed for the ticket reservation scenario. The target of these policies is specified in Table 4.1. This table shows any policy request that is related to service invocation among all services with a given range of ports must be evaluated. This scenario is described in section 4.5.2. In the following, we will give a brief description of each policy.

Trust Policy. Trust policy is similar to the example given in Figure 4.4. This policy says that any access to a service with trust value of less than threshold (here, 0.5) will be denied.

TrustPause Policy. If the trust value of the target service is less than threshold, then it sends an obligation that asks the PME to pause the service for a certain amount of time (specified by TM service). This policy is useful for throttling requests to services.

TrustRedirect Policy. This policy is similar to TrustPause policy, except for redirecting the service call instead of pausing. Once TM service receives a XACML response with redirect obligation, it searches for the best alternative for the target service and sends the service address to the PME component. This policy is useful to replace a compromised service, instead of blocking the service call.

AirlineBlacklist Policy. This policy blacklists a set of airline services (here, AA0–AA9 services). Therefore, all requests targeted to these services will be denied.

AirlineWhitelist Policy. This policy specifies a list of services as a whitelist. Access to any other services would be denied.

AirlineTrust Policy. This policy combines the trust policy withe the airline policy. The policy in the table shows all access to AA0–AA9 services will be denied if they have a trust value less than threshold.

Price Policy. This policy blocks any ticket reservation request for a ticket with value higher than 600. To enable this policy, PME component extracts the value of ticket and encapsulates it in the metadata. Once TM receives this request, extracts this value and creates a suitable XACML request with price attribute. This policy is useful to prevent accidental service requests or to enforce corporate policies.

PriceTrust Policy. This policy is similar to previous policy except for checking the trust value too. If the price is higher than a threshold and trust value of the target service is less than trust threshold, the request will be denied.

PriceTrustDate Policy. This policy demonstrates how multiple policies can be combined to create more complex policies. This policy extends the PriceTrust policy by checking the ticket date too. If the request is for reserving a ticket in a range of specified dates, the trust value of the target service is lower than a threshold, and the price of the ticket is more than $600, then it blocks the request. Otherwise, the operation proceeds.

Table 4.1: Targets of the XACML policies.

Subject	localhost:90[0-9][0-9]
Resource	localhost:90[0-9][0-9]
Action	INVOKE

Table 4.2: XACML policies for the ticket reservation case study.

Policy	**Condition**	**Effect**
Trust	Env (`http://cs.purdue.edu/soa/trust`) <0.5	Deny
TrustPause	Env (`http://cs.purdue.edu/soa/trust`) <0.5	Pause
TrustRedirect	Env (`http://cs.purdue.edu/soa/trust`) <0.5	Redirect
AirlineBlacklist	Env (`http://cs.purdue.edu/soa/al`) == `AA[0-9]`	Deny
AirlineWhitelist	Env (`http://cs.purdue.edu/soa/al`) != `AA[0-9]`	Deny
AirlineTrust	Env (`http://cs.purdue.edu/soa/al`) == `AA[0-9]` && Env (`http://cs.purdue.edu/soa/trust`) <0.5	Deny
Price	Env (`http://cs.purdue.edu/soa/price`) >600	Deny
PriceTrust	Env (`http://cs.purdue.edu/soa/trust`) <0.5 && Env (`http://cs.purdue.edu/soa/price`) >600	Block
PriceTrustDate	Env (`http://cs.purdue.edu/soa/trust`) <0.5 && Env (`http://cs.purdue.edu/soa/price`) >600 && 2015-09-11 <= Date <= 2015-09-12	Block

4.5 Evaluation

In this section we first measure the performance overhead of the different types of AOP advices. Next, we describe the ticket reservation case study, which will be used throughout this dissertation. Finally, we demonstrate the performance and security effectiveness of the proposed PME framework through two different EC2 cloud testbeds.

4.5.1 AOP Performance Overhead

In this section, we conducted a experimental study to understand the performance impact of the basic AOP abstractions regardless of the target operations (in contrast to the overhead of the whole proposed architecture, which will be presented later). The hardware testbed is an Apple Macbook Pro with a 2.6GHz Intel Core i7 processor and 16GB of RAM. In this experiment, we designed a simple banking application with credit and debit operations. For each row in Table 4.3, we activated the corresponding pointcut. To exclude the effect of the semantic of the program, we also measured the performance of the same operations without AOP. Each execution calls the debit method for one billion times. The execution of the program without any AOP pointcut takes 581msec (0.581 nano seconds for each operation). As Table 4.3 shows, the overhead of single AOP interceptions are (less than 100nsec for every interception). Therefore, whenever the interceptions in an AOP-based application are not too frequent, the overhead of AOP framework could be very low compared to the expensive network operations in web service and SOA scenarios, which are usually in the order of milliseconds.

Table 4.3: Performance overhead of AOP advices using JBoss AOP.

Advice Target	**Advice Type**	**With AOP (msec)**	**Single Interception (nsec)**
Method	around	62854	62.9
	before	21658	21.6
	after	21218	21.2
Constructor	around	75162	75.2
	before	34215	34.2
	after	34548	34.5
Field	set	42277	42.3
	get	38154	38.2

4.5.2 Ticket Reservation Case Study

In this section, we present a ticket reservation case study, which will be used for the experiments throughout this dissertation. This SOA application, as shown in Figure 4.5, is composed of three kinds of services: ticket reservation services (denoted by TR), airline services (denoted by AA, DL, and UN[1]), and banking services (denoted by BN). In this application, client invokes a ticket reservation service (TR_1 or TR_2) to search for a flight and receive a list of relevant airfares. The search request is a JSON serialization of a `SearchReq(origin, departure, date)` object. To perform this operation, the TR service invokes three airline services to get the available airfares from each service. In this scenario, we assume there are three airlines and each airline is represented by three equivalent airline services provided by different companies. In Figure 4.5, AA stands for an American Airline service (which provides relevant airfares from an American airline), DL stands for a Delta airline, and UN stands for a United airline service. The TR service assembles all the received airfares from three airline services and send a response back to the client. In the next operation, user can choose an airfare from the list of returned airfares and initiate a confirmation operation. The reservation request is a JSON serialization

[1]We may refer to any of the airline services as AL service.

of a `ReservationReq(origin, departure, date, airline, price)` object. Once TR service received a confirmation request, it forwards it to a corresponding airline service and the target airline service communicates with a banking service (specified by BN) to finalize the payment.

Figure 4.5 shows the scenario after a successful search and an airfare confirmation operations. Each edge weight represents the number of service invocations happened between a pair of services so far. In this scenario, TR_1 has initially called AA_1, DL_1, and UN_1 for the search operation. In the next step, client asks the TR_1 to confirm a ticket from UN_1, which in turn it calls the BN_1 banking service. It is worth mentioning that this diagram is a screenshot taken directly from the UI. To simplify the figure, the policy engine and the trust manager service are not shown. However, we have deployed the PME components in all service domains that enable monitoring and enforcement of the service interactions.

In this Figure, C_i represents a class of service (CoS). All classes in a CoS have similar functionalities and could be used interchangeably. The CoS is useful in redirection and dynamic service reconfiguration scenarios. In the redirect scenario, one of the services in the same CoS with the highest trust will be chosen. We will discuss the dynamic service composition scenario in chapter 6. All services in the ticket reservation case study are implemented as REST services using JAX-RS technology and Jersey framework.

4.5.3 Performance Evaluation

We conducted two experiments in the Amazon EC2 cloud computing infrastructure to verify the effectiveness and performance of the PME framework.

Goal of the experiment. The goal of this experiment is twofold. First, we want to measure the performance overhead and scalability of the proposed PME framework. For performance evaluation, we measure the effect of this framework on the response-time and throughput of the ticket reservation application. Second, we want to verify

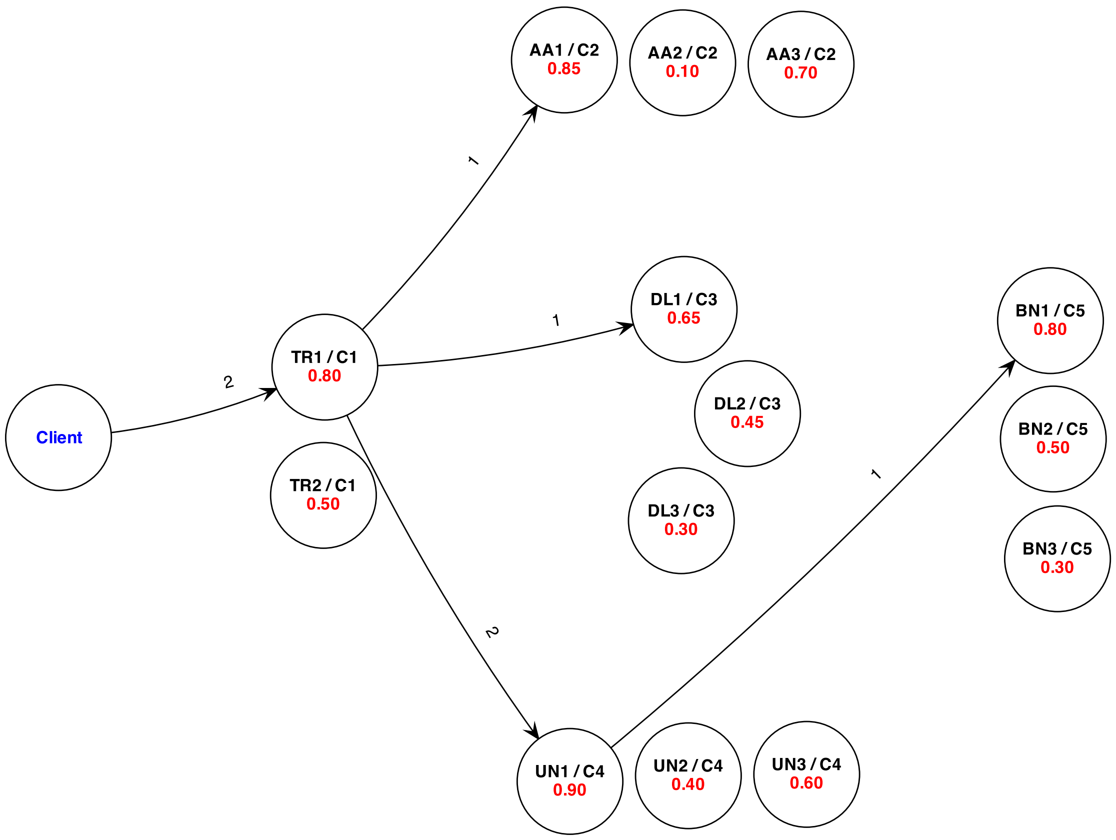


Figure 4.5.: The ticket reservation case study.

that the PME framework improves the security of the system. There are two ways for improving the security of a system, taking preventive actions and taking corrective actions. For the first approach, we make sure that the proposed framework is able to monitor the policies presented in Table 4.2 and prevent malicious activities. For the corrective approach, we measure the trust enhancement quantitatively achieved by taking corrective action (redirection).

Method of the experiment. To verify the scalability of the proposed solution, we conduct experiments in two testbeds with different computational powers in the Amazon elastic cloud computing (EC2) infrastructure. The details of the first testbed is presented in Table 4.4. In this testbed, we run the services in virtual machines with limited CPU, memory, and storage capabilities. To minimize the effect of variable delays in the client calls, the client (Apache Benchmark [62]) is also deployed in another virtual machine for both experiments. Therefore, all participating components (all application services, TM service, policy engine, and client) are deployed in the cloud. The second testbed we increase the capabilities of the virtual machines as presented in Table 4.5.

Table 4.4: Specifications of the first Amazon cloud testbed.

Service instance	t2.small (1 vCPU, 2GB memory, and EBS storage)
PE and TM instances	t2.small (1 vCPU, 2GB memory, and EBS storage)
Client instance	t2.micro (1 vCPU, 1GB memory, and EBS storage)
Operating system	Amazon Linux 2015.03 64-bit OS
Geographical region	US-w2 (Oregon region)

Table 4.5: Specifications of the second Amazon cloud testbed.

Service instance	c3.xlarge (4 vCPU, 7.5GB memory, and SSD storage)
PE and TM instances	c3.xlarge (4 vCPU, 7.5GB memory, and SSD storage)
Client instance	t2.micro (1 vCPU, 1GB memory, and EBS storage)
Operating system	Amazon Linux 2015.03 64-bit OS
Geographical region	US-w2 (Oregon region)

Input parameters of the experiment. For the performance measurement, we use the Apache Benchmark [62] tool to generate requests with varying levels of concurrency. In each run, we sent 1000 requests to the TR service. The number of concurrent requests sent to the ticket reservation service (TR) was varied from 1 to 8. Therefore, if the level of concurrency is 4, the Apache Benchmark keeps 4 number of active concurrent requests at any given time during the runtime of the experiment. Once one of the requests is finished, it replaces it with a new request. This scenario measures the performance of the confirmation operation (chain of $client \rightarrow TR \rightarrow AL \rightarrow BN$).

Results, analysis, and conclusions of the experiment. Figure 4.6 compares the average response times of the ticket reservation case study for three scenarios deployed in a testbed one (specifications in Table4.4). The three scenarios are baseline (with no security measure), monitoring (the PME framework without enforcement), and enforcement (the complete PME framework with both monitoring and enforcement activated). The first observation is that the response time for each scenario increases linearly according to the load. The reason is that in this testbed there is only 1 vCPU and all extra loads need to wait to get CPU time for processing. For 4 and 8 concurrent requests, there is a small waiting time added to the processing time. For example, once we increase the concurrency of a baseline from 1 to 4, the average response time increases from 133ms to 551ms, which shows 19ms of waiting time ($551 = 4 \times 133 + 19$). The second observation is that the enforcement scenario has a noticeable overhead compared to the monitoring scenario. This difference is caused by the blocking calls made by PME component to the TM service in the enforcement scenario. These calls triggers the invocation of policy engine by TM service, which adds to the response time. On the other hand, the session feedback in monitoring scenario is simple and asynchronous that happens in another thread without blocking the execution of the service.

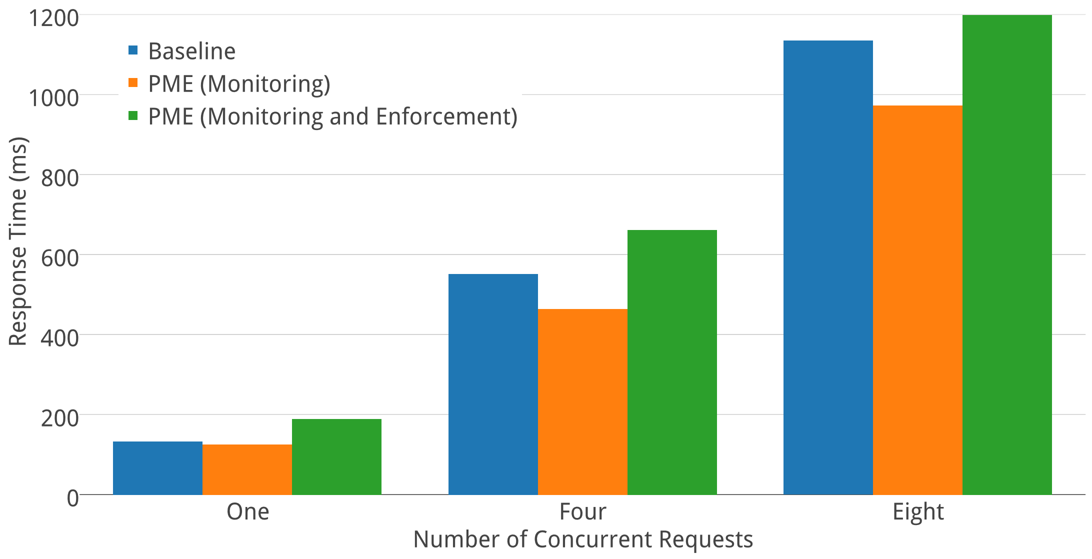


Figure 4.6.: Amazon EC2 testbed 1: Response-time of three scenarios (baseline, monitoring, and monitoring with enforcement) under multiple concurrency-levels for the ticket reservation case study.

Another interesting observation is that the average response time of monitoring scenario is less than the average response time of baseline scenario, which looks unexpected. To investigate the root cause of this observation, we further looked at the execution times of the TR service for all requests. Figure 4.7 shows the response times for both baseline and monitoring scenarios for concurrency level 4. This figure shows that the response times for the first 100 requests in the monitoring scenario are significantly higher than the baseline. After around 100 requests, it starts decreasing until around 200 and then it roughly becomes stable. We also observed a similar pattern for the response times of the AL service too. This observation can be explained by understanding the way AOP framework and JVM HotSpot virtual machine work. At the low level, the AOP framework operates by rewriting the Java bytecodes of a class at runtime once that class is loaded for the first time. Therefore, AOP framework automatically instruments every class on the fly whenever they are going to be used for the first time. This operation leads to a slower initial execution. However, the subsequent invocations of the same class will use the already generated bytecodes and their execution time will only depend on the complexity of the woven AOP aspects. This mechanism explains the slow initial invocations, but does not explain why the monitoring bytecodes should eventually run faster than the baseline bytecode. To understand it, we need to understand the way JVM HotSpot works. The official Java documentation explains the HotSpot as follows [63]:

"It includes dynamic compilers that adaptively compile Java bytecodes into optimized machine instructions..."

This paragraph explains that the actual machine code that are executing in the CPU may adaptively change over time based on HotSpot's criteria. Therefore, it shows that the bytecode generated by the AOP framework are efficient and they likely trigger the advanced optimizations at the HotSpot.

Another interesting observation in Figure 4.7 is that it shows a few spikes at around requests 200, 643, and 773. These spikes could be attributed to the random changes in the available communication and computation resources for the testbed's virtual machines as these virtual machines are collocated with a large number of virtual machines from other customers.

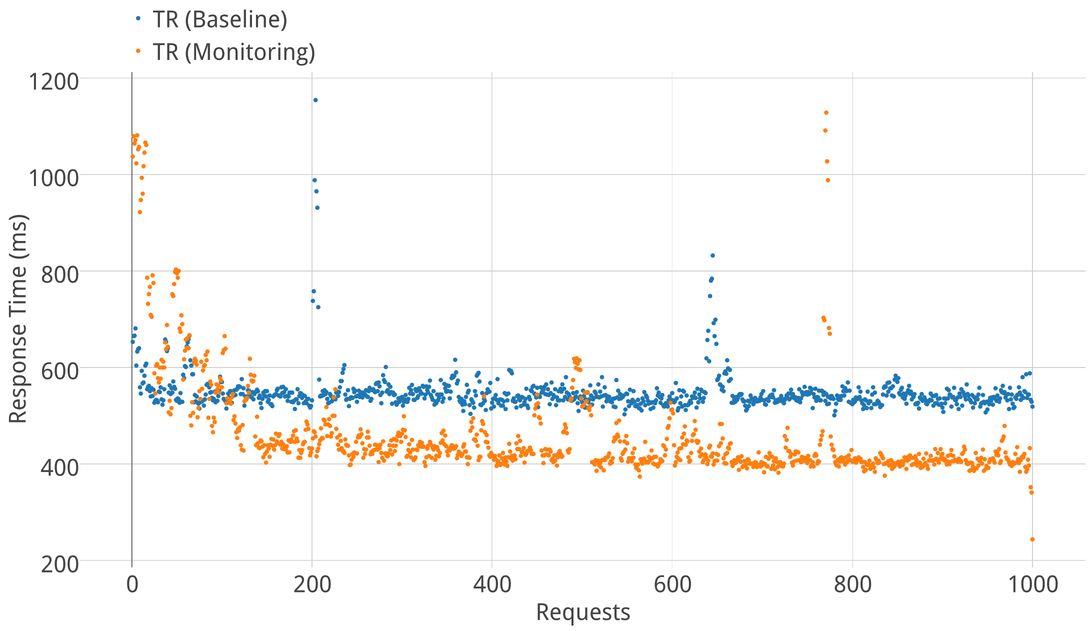


Figure 4.7.: Response-time of the TR service for baseline and monitoring scenarios for all requests in testbed 1.

These results presented in Figure 4.7 confirm that the PME framework has no overhead in the monitoring mode and it has a low overhead for the enforcement scenario (19.9% for the concurrency level 4 and 6% for concurrency level 8).

Figure 4.8 shows the result for the previous ticket reservation case study in a more powerful testbed (parameters presented in Table 4.4). Under concurrency level 1, the improvement in runtime execution of all scenarios is significant (roughly around 29% of improvement) but not linear to the number of processors. The reason is the extra vCPUs are useful when there are concurrent executions threads to take advantage of it. However, in this scenario, the extra vCPUs are underutilized (except for the system and internal Jetty threads that are not significant factors). The main factors in the observed performance improvement (in the concurrency level 1) are larger main memory and SSD storage. Another interesting observation is that the average response time for the monitoring scenario is no longer smaller than the baseline.

Figure 4.9 shows the response times for all requests similar to Figure 4.7 except for running on a faster testbed. Interestingly, after the initial 100 requests, most of the response times converge toward the average which is almost similar to both monitoring and baseline scenarios. In this testbed, the HotSpot virtual machine has triggered the advanced optimization to both scenarios to take advantage of the available resources. We observed a similar pattern for the other concurrency levels too.

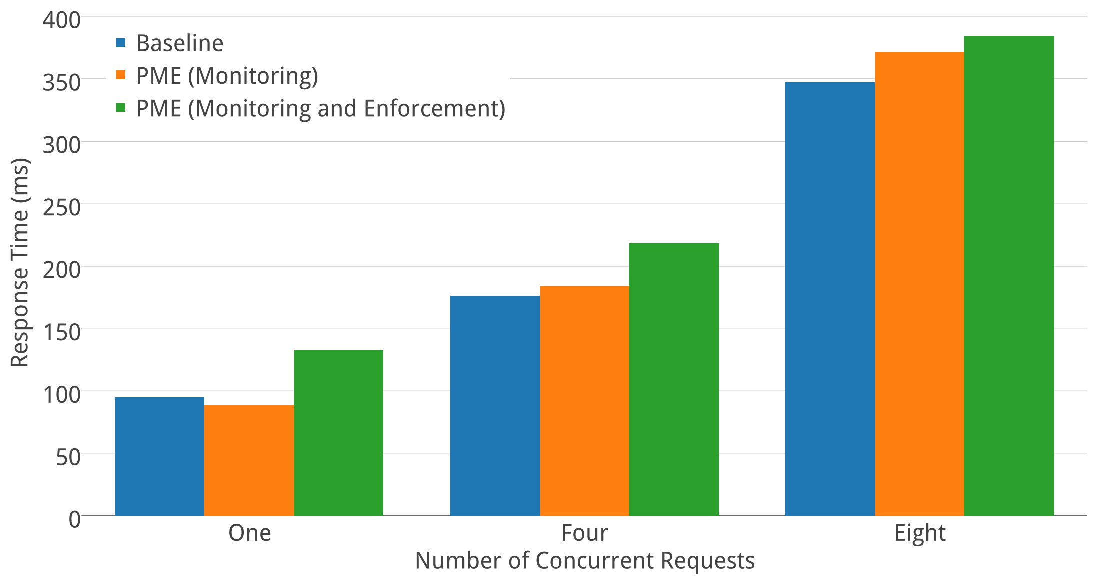


Figure 4.8.: Amazon EC2 testbed 2: Response-time of three scenarios (baseline, monitoring, and monitoring with enforcement) under multiple concurrency-levels for the ticket reservation case study.

On the other hand, the second testbed demonstrates its capability for the higher concurrency levels. For the concurrency-level 4, the response time is around 60-68% less than the first testbed in Figure 4.6. The improvement is slightly better for the concurrency level 8 (around 62-69% improvement), but since the number of vCPUs are 4, the improvement is not significant.

It is worth noting that response time is only one aspect of the performance evaluation. Another metric that complements the response time is throughput, which represents the average number of served request per seconds. Figure 4.10 shows that

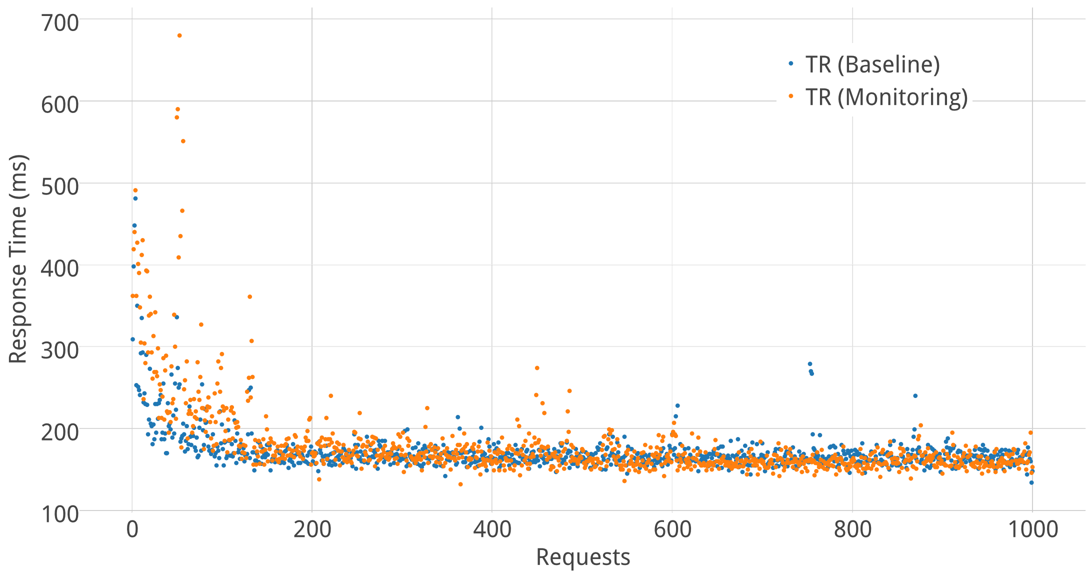


Figure 4.9.: Response-time of the TR service for baseline and monitoring scenarios for all requests in testbed 2.

unlike the response time, the throughput of baseline and monitoring experiments are fairly stable and do not change significantly based on different concurrency levels. However, the throughput of the enforcement scenario increases as we increase the concurrency level (26% increase from level 1 to level 8). The reason is that in this case even though each request receives less CPU time, however, the overall utilization of the CPU is higher than in a single concurrency level scenario as the requests are getting blocked during the invocation of the TM service. But, if we exclude the TM service call, as in the baseline scenario and monitoring scenario level 4 to level 8, the overhead of context switching between threads takes over and the overall throughput decreases slightly.

Figure 4.10 shows the throughput of system for the second cloud testbed. This diagram shows the throughput of all three scenarios increases around 116–145%. The improvement is more noticeable for the enforcement scenario which as the throughput gap drops to less than 10% compared to the baseline scenario.

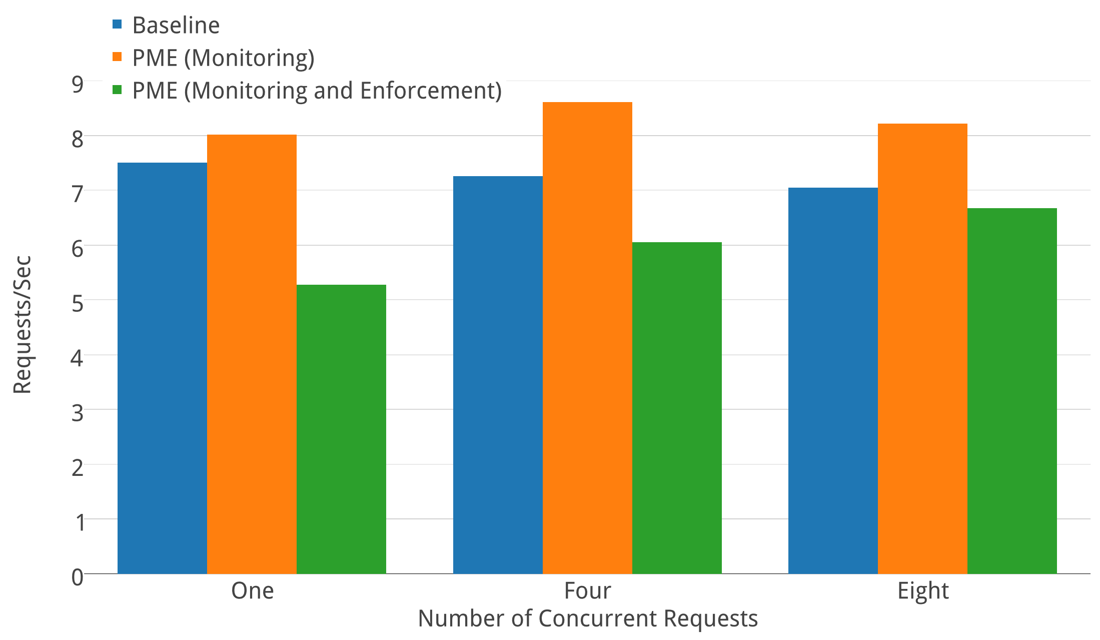


Figure 4.10.: Amazon EC2 testbed 1: Throughput of three scenarios (baseline, monitoring, and monitoring with enforcement) under multiple concurrency-levels for the ticket reservation case study.

In this part we briefly discuss our observations for the distribution of the response times, the request failure rate, the communication overhead, the effect of composite trust algorithms, and finally policy overhead:

- *Distribution of the response times.* We further studies the distribution of response times in both testbeds. The results for the concurrency-level 1 are presented in Figure 4.12. This figure shows that the difference between response times before the 80%-percentile is not significant. Therefore, less than 20% of requests are responsible for the most of the difference between average response times of all three scenarios. Interestingly, the major increase happens at the 10% (or even at the top 1%) of the requests, which mostly happen at the beginning of the experiment, which both AOP framework and HotSpot virtual machine apply advanced bytecode rewriting and optimizations. The other root cause of this observation is a random fluctuation in the available computational resources and network delays in the cloud that happen every once in a while.

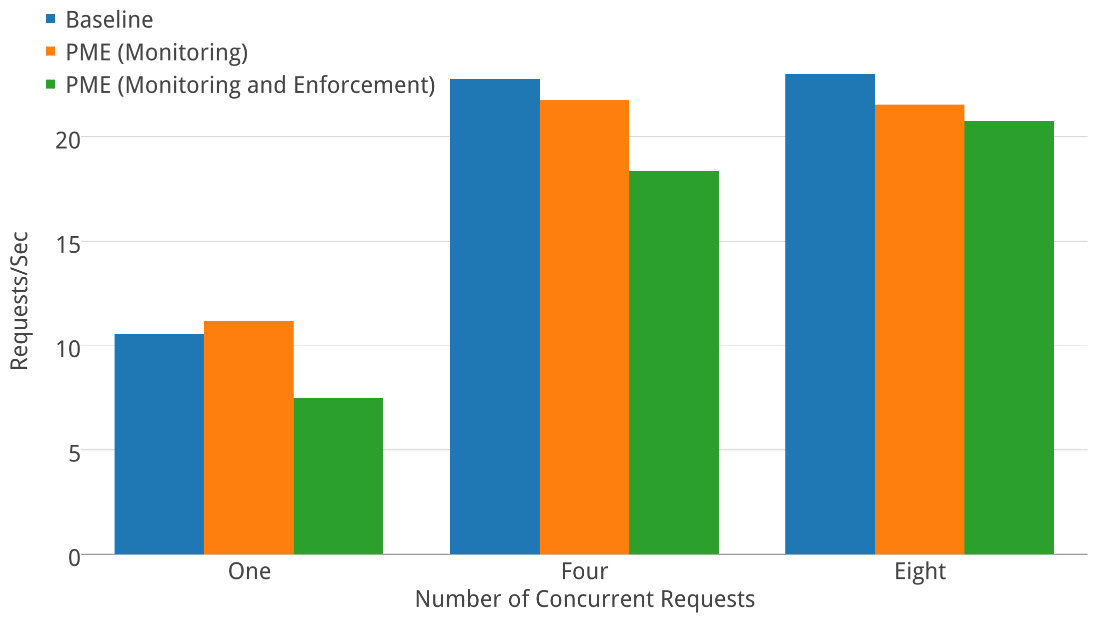


Figure 4.11.: Amazon EC2 testbed 2: Throughput of three scenarios (baseline, monitoring, and monitoring with enforcement) under multiple concurrency-levels for the ticket reservation case study.

But, as we explained this is not a major factor. This observations shows that if we conduct the measurement after a warm up period (could happen by some random traffic) can improved the perceived performance by users.

- *Request failure rate.* In these experiments, the failure rate for the requests was 0% and all requests were completed successfully.
- *Effect of composite trust algorithms.* In this experiment, we used the coarse-grained global algorithm as the composite trust algorithm in the TM service. We further conducted an experiment to analyze the effect of the alternative graph-based algorithm. The overhead of using the graph-based algorithm was consistently less than 2ms. This result is not surprising, as the complexity of the graph-based algorithm is $O(|V|+|A|)$ and it only becomes more significant in extremely large service graphs.
- *Communication overhead.* The communication overhead (connection time) between virtual machines in the same geographical region is very low. The mean

overhead was 1 with standard deviation of between 0.3–0.8. It means for majority of the requests the connection time was between 1–2ms (except for a few spikes as discussed in Figure 4.7).

- *Effect of policies.* We also conducted experiments to measure the effect of policies (presented in Table 4.2) on the overall performance. The difference between response time overhead of these policies was less than 3ms (`PriceTrustDate` had the highest response time).

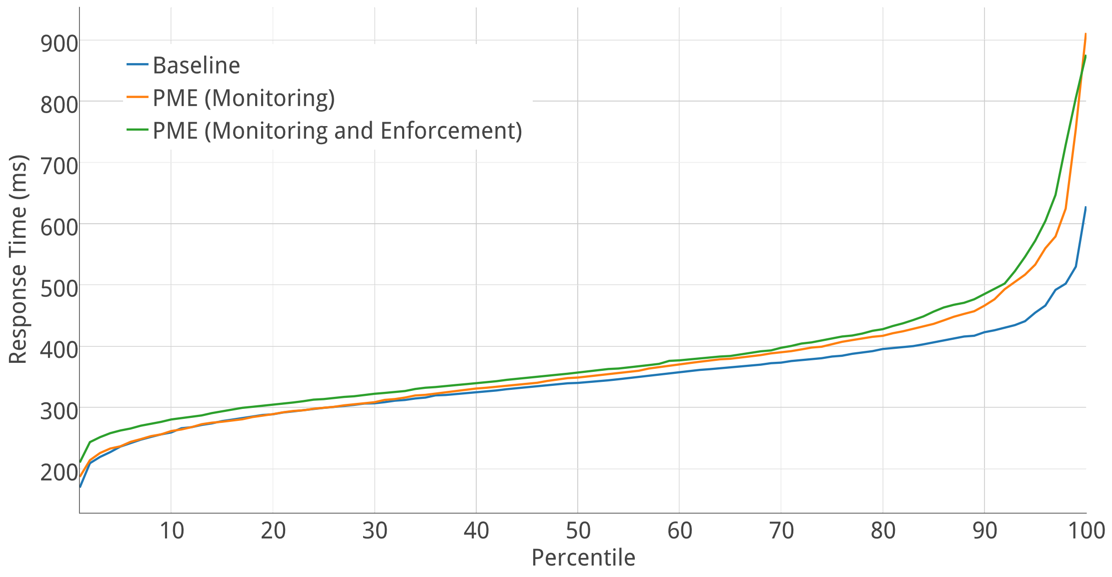


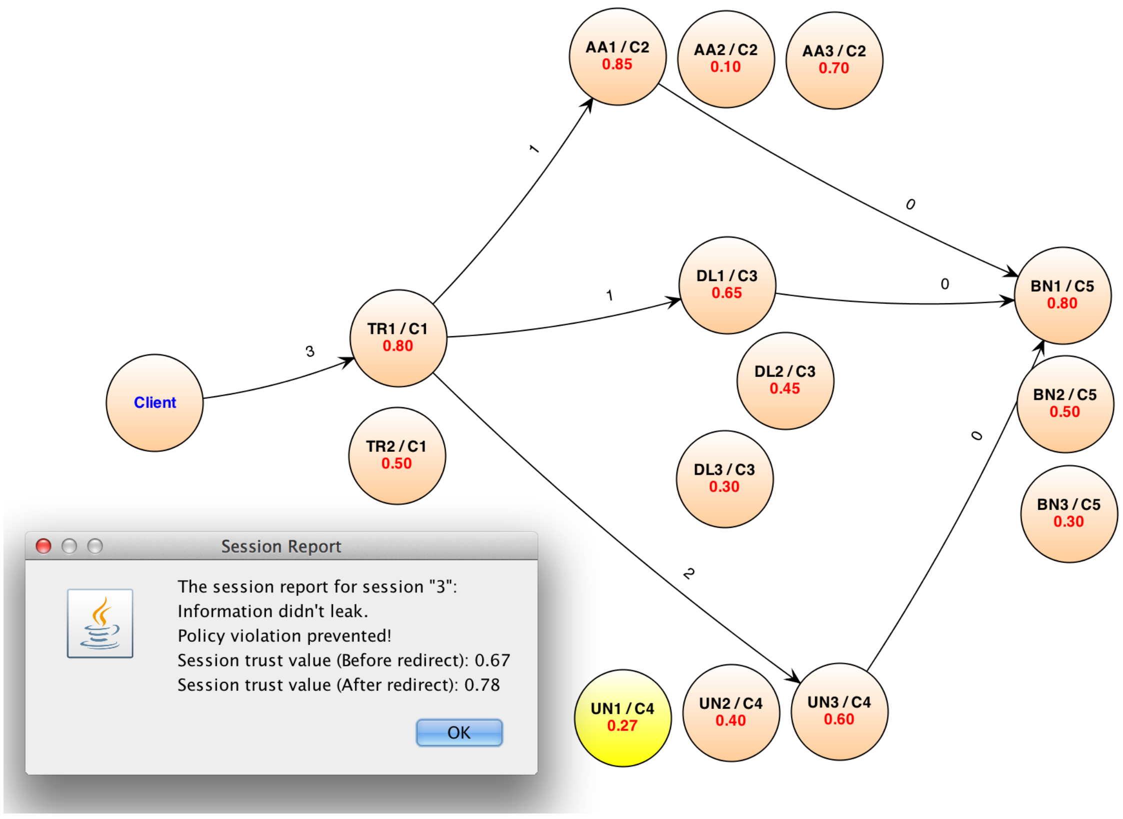
Figure 4.12.: Percentage of the requests served within a certain time (in milliseconds) for testbed 2 with concurrency-level 8.

4.5.4 Security Evaluation

For the security validation, we have two inputs. One of the is the type of policy, which is going to be monitored and enforced. The other source of input is changing the trust value of a service, which simulates a service compromisation attack. We will discuss the security validation in the next subsection.

In addition to the performance evaluation experiments which were discussed earlier, we also evaluated the security measures provided by the PME framework on multiple service graphs. In this section, we briefly discuss them using the ticket reservation case study presented in Figure 4.5. We approached the security evaluation in two ways. In the first way, we verified that the PME framework takes the specified security measures correctly under the malicious conditions. Second, we measured the trust improvement caused by taken security actions. There are two categories of security actions. Some of them are preventive, which prevent the malicious activities. The second category try to correct those activities. In this dissertation we provide solutions from both category. For the first category, we conducted extensive experiments based on the policies presented in Table 4.2 using the generic scenario generation feature of the UI. All those policies that have the Deny or Block effects have successfully prevented the malicious service invocations. In this dissertation, we provided two corrective security measures, redirection (which we will discuss it here) and ASSC (adaptive secure service composition), which will be discussed in chapter 6. The increase in the trust level caused by both of them are available through the security report of the UI. Figure 4.13 is an screenshot of the session report, that shows the effect of dropping the trust level of UN_1 service from 0.90 to 0.27 (bellow trust threshold), while the `TrustRedirect` policy is enabled. This transaction has happened right after the execution state presented in Figure 4.5. In this scenario, instead of blocking the confirmation, the PME framework replaces the untrusted service with a more trustworthy service in the same category (which is UN_3 with trust value of 0.60). The session report shows that the session trust has increased from 0.67 to 0.78 by redirecting the request to UN_3.

The experiments presented in this section, confirm that the PME framework is practical and effective. The experimental results shows that the presented framework is scalable and it is able to takes advantage of modern multi-core processors.



The session report for session "3":
Information didn't leak.
Policy violation prevented!
Session trust value (Before redirect): 0.67
Session trust value (After redirect): 0.78

Figure 4.13.: Enforcement of redirect policy in the ticket reservation case study.

4.6 Related Work

AOP was originally introduced in [64, 65]. This paradigm extends the object-oriented programming by enabling the definition of independent crosscutting concerns. De Win et al. [66] pioneered the use of AOP in the security domain. However, they did not address the policy monitoring or enforcement. Shah and Hill [67] extend the C language to support AOP and apply it on traditional applications to address the security issues. However, they only give high-level guidelines without discussing the technical details.

The runtime monitoring of web services has been the focus of many research efforts. Li et al. [68] describe a system for auditing runtime interaction behavior of web services. They address this problem by using finite state automata to validate the predefined interaction constraints. [69] presents a comprehensive auditing solution for verifying the correctness of web service interactions through creating an automata and examining the *safety* and *liveness* properties of the application. The weakness of such proposals that require a formal modeling is the fact that they are suitable for complex real world scenarios as they expect precise modeling of services.

There are a number of web service QoS monitoring frameworks based on AOP such as [70–74]. Compared to our proposal, they are based on message interceptions and use specific orchestration technologies (e.g., BPEL [75]) and they do not address the security policies. [76] proposes a self-healing solution for BPEL by extending the ActiveBPEL engine. They define two domain-specific languages to define the corresponding rules. However, their solution is technology-dependent and these rules cannot be modified at runtime.

Policy enforcement is investigated through different perspectives. There are two main approaches to the policy enforcement. The first approach is through ensuring all services behave as specified. In this category formal modeling techniques are used to model the applications. Next, we formally prove that the application code is secure and will not violate any policies. This category is highly abstract and cannot be easily adapted to different use cases of real-world applications as modeling them could be very difficult. The second approach is to ensure that services do not behave in a malicious way through monitoring potentially suspicious activities and preventing unauthorized activities. In this dissertation, we chose the second approach as it gives a flexibility and ease of use for real-world scenarios. In the following, we discuss a few research work related to policy enforcement.

An example of the first enforcement category is AVANTSSAR platform [77], which defines a language and a few tools to formally define and automate the validation of trust and security in SOA through model checking and reasoning techniques. However, these techniques are static and they require time-consuming modeling and system specification. Moreover, they require access to the source code of the services to be able to analyze them. On the other hand, our enforcement solution is a dynamic approach and does not require any access to the source code of the services beforehand. Moreover, a large number of research activities have been focused on techniques based on static program rewriting [54, 78, 79], which suffer from the similar weaknesses.

[80] presents an enhanced Enterprise Message Bus (called xESB), to enforce the access control policies in SOA. The shortcoming of their approach is that they assume SOA applications are based on the ESB architecture, which is not true in majority of the real world SOA applications. For example, majority of real-world service-based solutions (for example, Amazon SOA architecture and Netflix microservice-based architecture) use a subset of SOA properties without depending on complex service buses. Moreover, they present a non-standard policy language. Finally, they do not provide any concrete real-world case studies to evaluate their architecture.

Authors in [81] propose an solution for enforcing usage control policies. The limitations of this work are as follows. First, they do not provide dynamic trust establishment and maintenance. Second, their enforcing components are tightly coupled to the business logic of the services, which requires a considerable change in the services.

4.7 Conclusion

In this chapter, we designed and evaluated the inter-service PME framework, which non-intrusively monitors the execution of services at runtime and enforces the policies. To demonstrate the effectiveness and performance of this framework, we designed a ticket reservation case study and evaluated it on two different EC2 cloud testbeds (with different computation and memory power) under various scenarios. As a part of this evaluation, we designed nine realistic XACML policies that cover blacklisting, whitelisting, time-sensitivity, and application-dependent parameters (e.g., price in the ticket reservation case study). We implemented the PME framework using AOP. We further improved the implementation by parallelizing multiple operations of the system, for example, sending the monitoring session feedbacks to the trust manager asynchronously.

One of our observations is related to the way AOP works. Since AOP instruments the relevant Java classes once they are loaded into the system for the first time, the main overhead of the PME framework is observed at the beginning of its operation. Therefore, after a short warm-up period, the performance overhead of the PME frameworks will decrease significantly. For example, in the monitoring scenarios, the main overhead is related to the first 100 requests. After AOP instrumented all classes and Java JIT compiler (HotSpot) optimized the code for the target machine, the overhead is negligible (under 1%, while the average overhead for all operations is 3.5%). This is an interesting observation, as most of the real-world SOA deployments are long-running and the initial warm-up overhead of the PME framework will not be an issue. In production systems, we can warm-up the systems offline by some synthetic load before they start serving the customers' traffic.

In addition to response time, we also measured the throughput of the scenarios under similar testbeds. We observed that when the concurrency level of the input is equal to or higher than the number of available CPU cores, the multi-threaded framework uses the cores efficiently and the throughput remains stable. This condition always holds in all realistic scenarios.

Another observation is that the PME framework scales almost linearly with the number of available CPU cores. For example, once we increase the number of the CPU cores to 4, the response time of the services drops by 67%, which shows an 89% efficiency factor for the scaling (as the ideal decrease would be 75%). We further observed that the overhead of using the graph-based composite trust algorithm was consistently less than 2ms, and the overhead of evaluating the policies (for all nine policies) remained under 3ms for all the experiments. Moreover, we never observed any request failure under various loads, which indicates the robustness of the system.

One of the promises of the PME framework is to provide an end-to-end visibility to the clients. In our testbed, clients can query the trust manager after every session to get a report of that session. This report includes the potential policy violations and the quantitative improvement in the trustworthiness of the corresponding session by enforcing the corrective policies (e.g., through redirection). We further evaluated the security of the PME framework through extensive testing of system under various policies and observing that the framework is able to enforce the policies (by blocking a service, redirecting a service, or rearranging the whole SOA for an optimal composition).

5 INTRA-SERVICE POLICY MONITORING AND ENFORCEMENT

In this chapter, we propose a new fine-grained monitoring and enforcement framework for SOA, which is able to track the flow of sensitive information *inside* services. The structure of the chapter is as follows. First, we explore the motivations and requirements for designing an intra-service PME framework for SOA. Second, we design the proposed framework based on AOP and taint analysis and discuss the implementation details. Next, we demonstrate the effectiveness and practicality of the proposed framework through ticket reservation case study. Experimental results confirm that the approach is practical and effective. Finally, we discuss the future work and related work.

5.1 Motivation and Requirements

To provide high security assurance in SOA, it is not sufficient to model the services as black boxes and only inspect their inputs and outputs. Particularly, the black-box abstraction fails in scenarios where infrastructures and services are not under the direct control of clients (as in cloud computing).

The proposed PME framework in chapter 4 operates based on inspecting all interactions among services. This inter-service monitoring framework is able to discover the services that are either malicious or get compromised over time. Even though this technique is effective and efficient, however, it is not fine-grained enough to address the advanced incidents. The challenge is that a service may invoke another service without disclosing any sensitive information. Similarly, the untrustworthy results produced by this invocation may not be used to produce the final results for the client. In none of these cases, it is justifiable to punish a service by reducing its trust value.

On the other hand, the standard security techniques, such as firewalls and access control mechanisms, are not able to detect and prevent sophisticated information leakages [82]. In fact, it is beyond the scope of such mechanisms to determine whether information is used correctly inside a service. Information leakage is the root cause of privacy violation, which has been an enormous concern in the recent years.

Therefore, to guarantee strong protection of data confidentiality and integrity, there is a need for a fine-grained information flow tracking inside applications and services. To achieve this goal, we design a framework to monitor the internals of SOA components as well as interactions among them. Combining both frameworks would enable the security architecture to track how data is manipulated within each service and how this data is transferred among services.

Requirements The main requirements for an intra-service PME framework are as follows:

- *Minimal impact on the existing systems and transparency.* Most of the current taint analysis mechanisms requires changes in the Java runtime system or access to the operating system to be able to track the flow of information. These changes are not acceptable in majority of real world use cases. Therefore, a non-intrusive mechanism which requires minimal or no changes in the legacy systems is preferred. Furthermore, most approaches require access to the source code of services to analyze them and produce modified services. This requirement also prevents wide-spread adoption of the target intra-service framework.
- *Accuracy.* The flow tracking operation must be accurate to prevent unwanted false positive or false negative.
- *Runtime operation.* Static mechanisms are not suitable. We require a dynamic mechanism that is able to operate in runtime, in parallel with the actual service execution.

5.2 Design of an Intra-Service Monitoring and Enforcement Framework by AOP

We explain the operation of the intra-service monitoring framework through an example. We assume there is a scenario with a client C, the service under monitoring S_1, and another service S_2, which would be invoked by S_1. We assume client C sends a request to S_1, which includes sensitive PII (personally-identifiable information). Service S_1 stores the information in an object obj_c. In the next step, service S_1 invokes another service, S_2, by sending a request $Req_1 = f(obj_c, STATE_{s_1})$, which $STATE_{S_1}$ is a collection of all fields and accessible storage by S_1. Function f means that Req_1 is the outcome of a process that has involved obj_c and a subset of internal fields of S_1. If S_2 can recover the client's data C (partially or completely) from Req_1, then we say S_1 has leaked sensitive information to S_2. The goal of intra-service PME framework is to detect and prevent such scenarios.

Information flow control [82] is a mechanism to track the movement of information within a service or application. This mechanism labels the data sources from their origination and updates the labels according to the predefined propagation rules as data moves forward inside the program. Taint analysis is a form of information flow analysis. The goal of taint analysis is to keep track of some specific inputs during the execution of a program. These inputs are usually the entry point of sensitive information (e.g., users' private data) or untrusted data (untrusted inputs from users that would lead to attacks). In the context of this dissertation, we assume inputs are those methods in a service that receive users' private data that need to be protected against information leakage attacks. It is worth noting that even though taint analysis uses the *taint* as a potentially dangerous input to a program such as user inputs, however, since the operation of our algorithm is the same as taint analysis, we use the same terminology for tracking the sensitive information. Therefore, we will use the term *taint* for a sensitive information provided to a service and is expected to be protected from leakage.

The goal of this chapter is to design and implement an intra-service PME framework by using taint analysis. There are two main approaches to implement the taint analysis. The first approach is *static taint analysis* (e.g., TAJ [83]), which usually uses the program source code to reason about the information flow tracking. The second approach is called *dynamic taint analysis* (e.g., TaintCheck [84], Dyton [85], and DTA++ [86]), which infers the information flow by observing the execution of a program. The first approach cannot meet the first requirement listed in the previous section. Even though the second approach meets the first requirement, however, most of the available taint analysis schemes incur too much overhead (e.g., 10–25 times overhead in [84]) to be used in runtime. We are interested in a solution, which is efficient enough to be utilized in the production scenarios in runtime. To address these limitations, we decided to design a new taint analysis framework for SOA using AOP. Leveraging AOP enables designing a framework which is transparent to the services and users. Therefore, service providers are not required to change their services.

The taint analysis framework described in this section is implemented in Java using JBoss AOP [53] framework. In the following, we define the taint source, taint sink, and taint propagation policies in the proposed framework.

Taint sources. Generally, the taint source could be any entry point in the service, which receives or generates information that we are interested in tracking. The selection of taint sources depends on the context of the analysis. For example, taint source could be any information originated from a file, a network protocol, a keyboard, or mouse. We define the taint source as any information that a service received from a client or other services through a web service request. This request likely contains sensitive information or personally identifiable information (e.g., credit card numbers and social security numbers). Figure 5.1 shows the AOP pointcut that identifies the taint source in the ticket reservation scenario. In this pointcut, `maliciousReserve(..)` is an entry point method to the service. Once e request arrives to the TR service, the `processTRService()` advice from `edu.purdue.cs.soa.pme.TaintAction` aspect class will be called to mark this object as tainted. We keep a reference to all tainted objects in an `IdentityHashMap` data structure.

```
<!-- Taint source -->
<pointcut name="TaintSource" expr="execution(* edu.purdue.cs.soa.service.
    ticketservice.*->maliciousReserve(..))"/>
<bind pointcut="TaintSource">
    <around name="processTRService" aspect="edu.purdue.cs.soa.pme.TaintAction"/>
</bind>
```

Figure 5.1.: Taint source pointcut in ticket reservation scenario.

Using AOP, we can easily change the taint source definition to support any other taint sources depending on the scenario. Specifically, we may choose taint sources from APIs such related to file read, network access or input from library method calls. For example, the following methods could be other sources of taint:

- `HttpServletRequest.getParameter()` for Servlet based services.
- `PreparedStatement.executeQuery()` for input from SQL databases.
- `FileReader.read()` for reading from files.
- `System.getenv()` for reading environment variables.

Taint sinks. Taint sinks are defined as a set of execution points inside a service, where information may leave the service. For example, this information could be stored in a file or sent out to a network. We define the taint sink as any method that invokes another service. Figure 5.2 shows an AOP pointcut that identifies the taint sink in the ticket reservation scenario. This pointcut intercepts the calling of `post` method from jersey REST framework. Whenever the ticket reservation service calls this method within the project package, the `processTaintSink()` advice from `edu.purdue.cs.soa.pme.TaintAction` aspect class will be called to test whether any of its arguments (i.e., outgoing information) are tainted. In this case, it will notify the trust manager that an information breach has happened. Other sources of taint sinks are networking and I/O write APIs, such as:

```
<pointcut name="withinProject" expr="within(edu.purdue.cs.soa.ticketservice.rest
    .*)"/>

<!-- Taint sink -->
<pointcut name="TaintSink" expr="call(* com.sun.jersey.api.client.WebResource.
    Builder->post(..)) AND withinProject"/>
<bind pointcut="TaintSink">
    <around name="processTaintSink" aspect="edu.purdue.cs.soa.pme.TaintAction" />
</bind>
```

Figure 5.2.: Taint sink pointcuts for intra-service monitoring framework

- `HttpServletRequest.setParameter()` for Servlet based services.
- `FileReader.write()` for writing into files.
- `System.setenv()` for setting environment variables.

Taint propagation policies. If an instruction f uses a tainted object X to produce another object Y, then Y becomes tainted (or taint is propagated from object X to object Y). Taint propagation depends on the semantic of the operation. Taint propagation could be presented as an operator with transitive property:
$Y = f(X)$ and $Z = f(Y)$, then $Z = f(X)$.

We consider two categories of propagation policies. The first category addresses all method calls and field accesses that happen inside a service. We defined relevant pointcuts in the Figure 5.3. Lines 1–4 in this figure defines a pointcut for method execution within the target service. Whenever a method in this service is executed, the `processMethodExec()` method from from `edu.purdue.cs.soa.pme.MethodTracker` is called to process the taint propagation caused by this method call. This method extracts the arguments of the invoked method and uses the taint data structure to find whether any of them are tainted. The next step is to apply the other taint propagation policies inside the method body. If the return value is tainted, then the object that receives the output of this method will be tainted. Lines 6–9 processes the accesses to class fields. If the target field is tainted, the object that accesses this field will become tainted. Similarly, lines 11–14 process the assignments to class fields. If the value that is assigned to a field is already tainted, we add this field to a list of tainted objects. For all of the described operations, if the outcome is not tainted and is assigned to a tainted object, we will untaint that object by removing from the list of tainted objects.

```
<pointcut name="anyExecution" expr="execution(* edu.purdue.cs.soa.ticketservice.
    rest.*->*(..))"/>
<bind pointcut="anyExecution">
  <around name="processMethodExec" aspect="edu.purdue.cs.soa.pme.MethodTracker" />
</bind>

<pointcut name="fieldGet" expr="get(* edu.purdue.cs.soa.ticketservice.rest.*->*)"/
    >
<bind pointcut="fieldGet">
  <around name="processFieldGet" aspect="edu.purdue.cs.soa.pme.FieldTracker"/>
</bind>

<pointcut name="fieldSet" expr="set(* edu.purdue.cs.soa.ticketservice.rest.*->*)"/
    >
<bind pointcut="fieldSet">
  <around name="processFieldSet" aspect="edu.purdue.cs.soa.pme.FieldTracker"/>
</bind>
```

Figure 5.3.: Method pointcuts for intra-service monitoring framework

The second category of taint propagation policies, tracks the information flow in the operations that access multiple fields. Specifically, we are interested in String fields and the operations applied to them. Most of the sensitive information and PII data (e.g., credit card numbers, SSNs, and addresses) are represented by `String` objects. A selected list of these pointcuts are shown in Figure 5.4. In Java, String is represented by four different classes: `String`, `StringBuilder`, `CharArray`, and `StringBuffer`. Therefore, we define pointcuts to process all of these classes. For every string operation, we call respective method from the `edu.purdue.cs.soa.pme.StringTracker` aspect. Lines 23–25 shows the method for processing `constructor` operations. Similarly, lines 27-29 shows the processing of string comparison operation for different string classes.

However, the array fields require special consideration from AOP point of view. Therefore, we defined new pointcuts as demonstrated in Figure 5.5.

```
<pointcut name="withinProject" expr="within(edu.purdue.cs.soa.ticketservice.rest
    .*)"/>

<!-- String taint -->
<pointcut name="stringCtor" expr="call(java.lang.String->new(..)) AND
    withinProject"/>
<pointcut name="stringCtorModification" expr="(call(java.lang.String->new(byte[],
    int)) OR
          (java.lang.String->new(byte[], int, int)) OR
          call(java.lang.String->new(byte[], int, int, Charset)) OR
          call(java.lang.String->new(byte[], int, int, int)) OR
          call(java.lang.String->new(byte[], Charset)) OR
          call(java.lang.String->new(byte[], int, int, String)) OR
          call(java.lang.String->new(byte[], String)) OR
          call(java.lang.String->new(char[], int, int)) OR
          call(java.lang.String->new(int[], int, int))) AND withinProject"/>

<pointcut name="stringCompareTo" expr="call(public * java.lang.String->compareTo*(
    String)) AND withinProject"/>

<pointcut name="stringConcat" expr="call(* *->concat(..)) AND withinProject"/>

<pointcut name="stringBuilderCtor" expr="call(java.lang.StringBuffer->new(..)) AND
     withinProject"/>

<pointcut name="stringBuffReplace" expr="call(public * java.lang.StringBuffer->
    replace(..)) AND withinProject"/>

<bind pointcut="stringCtor OR stringBuilderCtor OR stringBufferCtor OR
    stringCopyValueOf OR stringFormat">
    <around name="processStringCtor" aspect="edu.purdue.cs.soa.pme.StringTracker"/
    >
</bind>

<bind pointcut="stringEquals OR stringCompareTo OR stringReplaceString OR
    stringContentEquals OR stringContains OR stringBuilderLastIndexOf OR
    stringBuilderIndexOf">
    <around name="processStringComparison" aspect="edu.purdue.cs.soa.pme.
    StringTracker"/>
</bind>
```

Figure 5.4.: String pointcuts for intra-service monitoring framework.

```
<!-- Tainted array field access -->
<arrayreplacement expr="class(edu.purdue.cs.soa.ticketservice.rest.*)"/>
<prepare expr="field(Object[] edu.purdue.cs.soa.ticketservice.rest.*->*)"/>

<interceptor class="aspect.ArrayInterceptor"/>
<arraybind type="READ_WRITE">
    <interceptor-ref name="FieldInterceptor"/>
        <advice name="processArray" aspect="edu.purdue.cs.soa.pme.TaintTracker" />
</arraybind>
```

Figure 5.5.: Array element access pointcuts for intra-service monitoring framework.

5.3 Evaluation

In this section, we evaluate the performance and effectiveness of the proposed intra-service PME framework by conducting experiments on ticket reservation case study (described in chapter 4 Figure 4.5) in the Amazon EC2 cloud computing infrastructure.

5.3.1 Performance Evaluation

Goal of the experiment. The goal of this experiment is to measure the performance overhead and scalability of the proposed intra-service PME framework, which is based on taint analysis. Similar to chapter 4, we conduct the performance evaluation in terms of response-time and throughput of the SOA application.

Method of the experiment. Similar to the experiments in chapter 4, we conduct the performance evaluation in two cloud testbeds. The first testbed is deployed based on the configuration presented in Table 4.4. In this experiment, we deployed the policy engine and TM service in a t2.small EC2 instance. We also deployed the ticket reservation services in another t2.small instance. To minimize the effect of unpredictable and variable delays on the client side, the client (Apache Benchmark [62]) is also deployed in a separate t2.micro virtual machine. For the second testbed, we deploys the services in more powerful virtual machines as described in Table 4.5. We have enabled the taint analysis framework in the TR services. We also enabled the PME framework in the AL services.

Input parameters of the experiment. In these experiments, we generate REST requests with varying levels of concurrency (1, 4, and 8). In each run, we sent 1000 requests to the TR service. This scenario measures the performance of the confirmation operation (chain of $client \rightarrow TR \rightarrow AL \rightarrow BN$), while the ticket reservation service is inspected by the taint analysis framework. The other source of input is policies. In these experiments, the TM service always first checks for the data leakage report. If any data leakage has happened, then it asks the PME components to block the data leakages without contacting the policy engine. If there is no data leakage, the TM service follows the same process as the PME framework in the enforcement mode.

Results, analysis, and conclusions of the experiment. Figure 5.6 compares the average response time of the ticket reservation scenario for the first Amazon EC2 testbed. These results confirm that the overhead of taint analysis framework is slightly higher than the PME framework in enforcement mode. In the taint analysis framework, the session feedbacks, which reports the service invocations to the TM service, are asynchronous and happen in separate threads. The amount of extra overhead is between 1.7–2.9%, which is caused by a more complex logic of the taint analysis aspects.

The overhead of both taint analysis and PME framework in enforcement mode compared to the baseline is due to synchronous calls to the TM service from all intermediary services. These calls triggers the invocation of policy engine and further processing in the TM service. The original service calls will be blocked during this process.

The effect of the concurrency-level on the response time is similar to the experiments in the previous chapter. significant. The main reason behind this delay is the fact that the concurrent service invocation causes each request to receive a less share of CPU. This problem is significantly higher in this testbed as the virtual machine has only access to a single vCPU. Therefore, all requests must wait longer as the number of concurrent requests increases.

Figure 5.7 shows the result for the same experiment run on the second testbed (parameters presented in Table 4.4). Under concurrency level 1, the improvement is noticeable but not significant. The reason is the extra vCPUs are mainly useful under a parallel workload and concurrent executions threads. However, in this scenario, the extra vCPUs are underutilized. For the concurrency-level 4, the response time for the taint analysis has improved around 67% compared to the first testbed in Figure 5.6. This speedup is very close to the ideal possible speedup (75% for 4 CPUs). The other observation is that since we only have access to 4 vCPUs, once we increase the concurrency level, the requests will be backlogged, which increases the response time significantly.

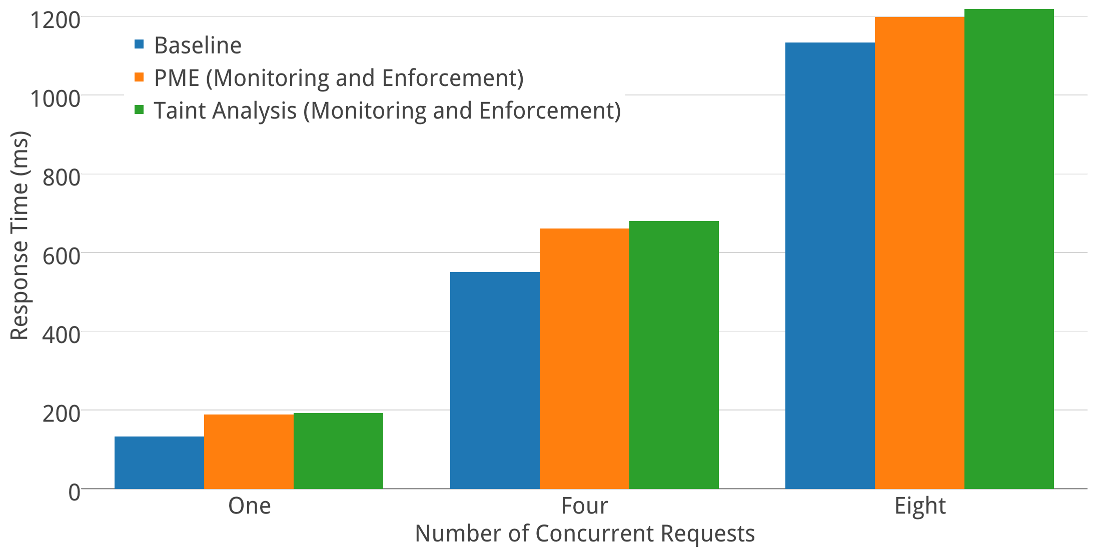


Figure 5.6.: Amazon EC2 testbed 1: Response-time of three scenarios (baseline, monitoring, and monitoring with enforcement) under multiple concurrency-levels for the ticket reservation case study.

Similar to the previous chapter, we also investigated the throughput of each testbed. Figure 5.8 shows that unlike the response time, the throughput of baseline

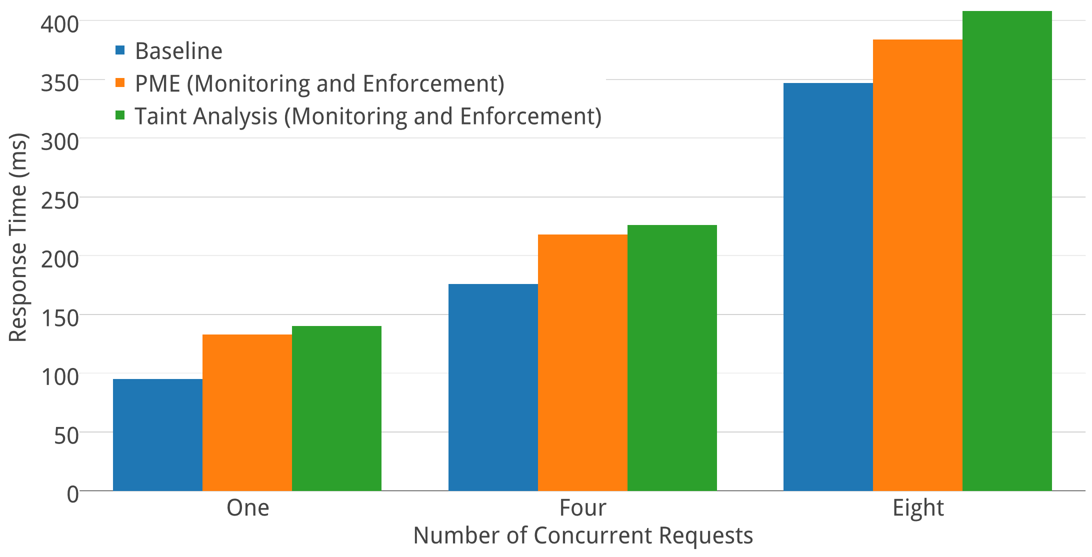


Figure 5.7.: Amazon EC2 testbed 2: Response-time of three scenarios (baseline, monitoring, and monitoring with enforcement) under multiple concurrency-levels for the ticket reservation case study.

and monitoring experiments are fairly stable and do not change significantly based on different concurrency levels. If we look at this figure more closely, the throughput increases slightly once we increase the concurrency level from 1 to 4. The reason is that in this case even though each request receives less CPU time, however, the overall utilization of the CPU is higher than a single concurrency level as the service block times during the TM service calls are used by other threads and requests are already pipelined in the Jetty framework and communication times are saved.

Figure 5.8 shows the throughput of system for the second cloud testbed. This diagram shows the throughput of taint analysis framework has increased 201% for the concurrency level 4 and 226% for concurrency level 8. These improvements confirm the scalability of the taint analysis framework as the number of CPU cores increases.

We also analyzed the distribution of response times, failure rate, communication overhead, and the effect of trust algorithms and policies. Since the results of these parameters were similar to the discussions in chapter 4, we do not repeat them again.

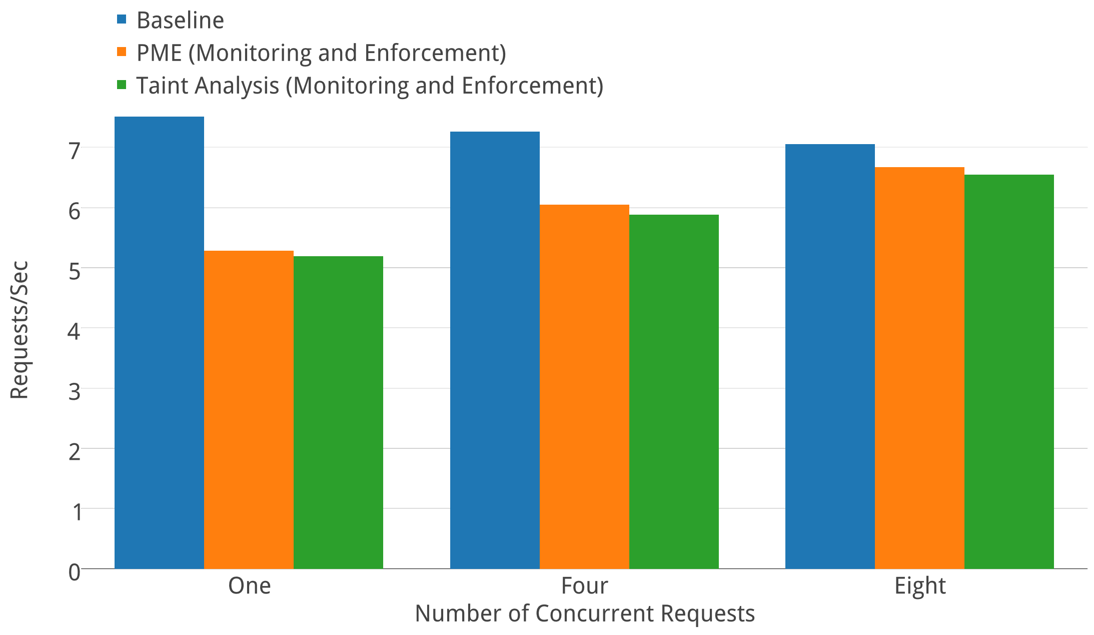


Figure 5.8.: Amazon EC2 testbed 1: Throughput of three scenarios (baseline, monitoring, and monitoring with enforcement) under multiple concurrency-levels for the ticket reservation case study.

5.3.2 Security Evaluation

In addition to the performance evaluation experiments, which were discussed earlier, we also evaluated the effectiveness of the taint analysis framework on multiple service graphs. In this section, we briefly discuss a data leakage attack scenario on the ticket reservation case study.

We have designed a data leakage attack which once it is activated, it invokes the malicious reservation REST API. Figure 5.10, shows a snippet of a malicious reservation service call in the ticket reservation service. This code manipulates the private data provided by client and sends it out to the airline service. The goal of this experiment is to verify that the taint analysis framework is able to detect and prevent such attacks efficiently.

The taint analysis framework is a preventive security measure. It means, it prevents the malicious activities instead of correcting them. In this attack scenario,

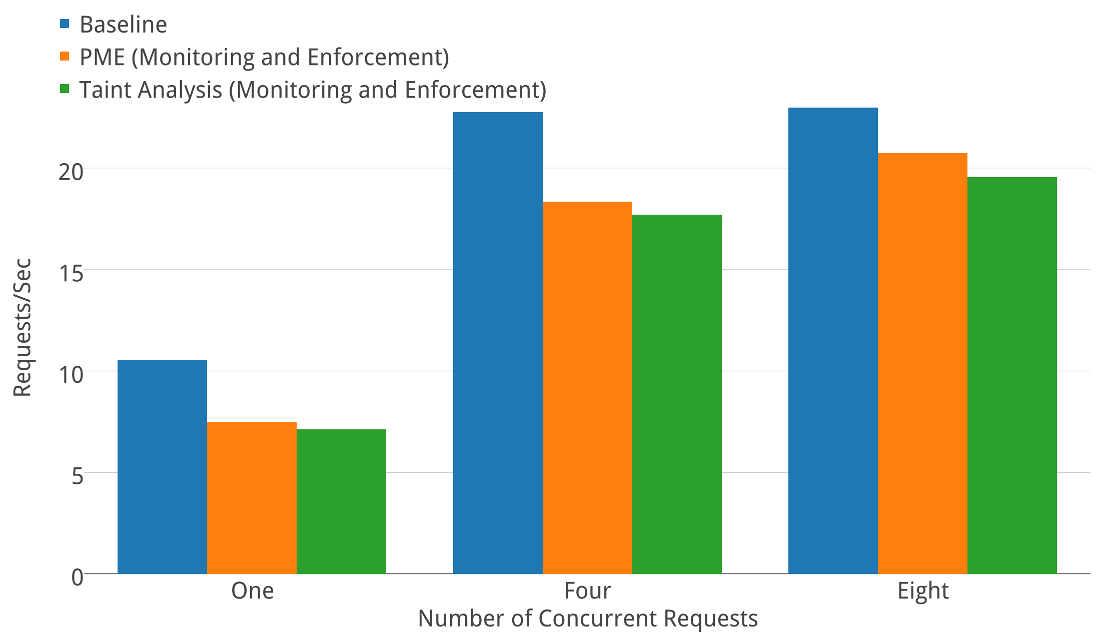


Figure 5.9.: Amazon EC2 testbed 2: Throughput of three scenarios (baseline, monitoring, and monitoring with enforcement) under multiple concurrency-levels for the ticket reservation case study.

client first searches for relevant airfares by querying the TR_1 service. Once it receives the airfares, it tries to confirm a ticket from UN_1. When the TR_1 receives the confirmation request, the taint analysis framework starts tracking the data provided by the client. For example, in Figure 5.10 snippet, the taint analysis framework labels the `privateInfo` as a sensitive data and tracks the propagation of it to other variables such as `subModifiedPrivateInfo`, `concatModifiedPrivateInfo`, and `modifiedPrivateInfo`. The propagations happen through calling String class APIs and invoking internal methods. Once the TR_1 service decides to call the airline service and send the `modifiedPrivateInfo` as data, the taint analysis blocks the calls and reports to the TM service that the TR_1 is going to leak the private information. When the TM service responds back with a *block* command, the taint analysis, terminates the current session. Figure 5.11 is an screenshot of the session report, that shows the effect of applying the data leakage attack on TR_1 service.

```
@POST
@Path("/maliciousreserve/{privateInfo}")
@Consumes("text/plain")
public String maliciousReserve(String req, @PathParam("privateInfo") String
privateInfo, @Context HttpHeaders headers) {
    ...
    subModifiedPrivateInfo = privateInfo.substring(0, 5);
    concatModifiedPrivateInfo = subModifiedPrivateInfo
        .concat(":extra-user-info").concat("#").concat(req);
    modifiedPrivateInfo = processPrivateInfo(concatModifiedPrivateInfo);
    ...
    WebResource webResource = client.resource(finalURL);
    ClientResponse response = webResource.accept("text/plain")
            .header("Authorization", headers.getRequestHeader("Authorization")
.get(0))
            .type("text/plain").post(ClientResponse.class, modifiedPrivateInfo
);
    ...
    String output = response.getEntity(String.class);
    return output;
}
```

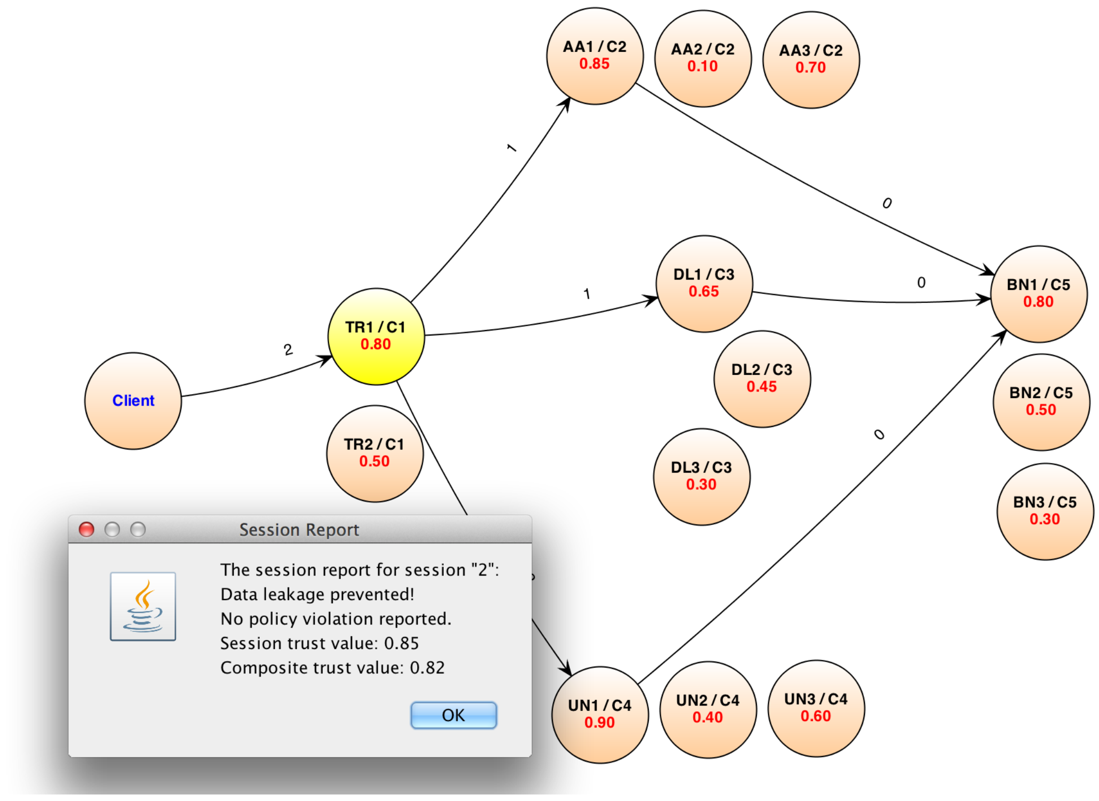
Figure 5.10.: A snippet of malicious reservation REST call, which leaks the client's private data.

As we mentioned before, once there is no data leakage attack in the service, the taint analysis enforces the client's policies. We also conducted extensive experiments based on the policies presented in Table 4.2 using the generic scenario generation feature of the UI. All those policies that have the Deny or Block effects have successfully prevented the malicious service invocations.

The experimental results verified that taint analysis framework is able to successfully identify the leakage of client's data to an external untrusted service.

5.4 Discussion

Improving the performance. Even though the performance overhead of the taint analysis framework presented in this chapter is much lower than some of the previous approaches (e.g., 10–25 times overhead in [84]), however, it still may be considerable



The session report for session "2":
Data leakage prevented!
No policy violation reported.
Session trust value: 0.85
Composite trust value: 0.82

Figure 5.11.: Screenshot of preventing the data leakage attack in the ticket reservation case study.

for non-critical real world production services. One solution to address this problem is to activate the intra-service mode only when it is necessary. For example, once inter-service PME framework detects suspicious activities by a service, or a user reports a low rating for a service, we can trigger the intra-service monitoring mode. Another solution is to scale-out the intra-service framework by allocating more resources.

Data-flow dependence vs control-flow dependence. Theoretically, the privacy and data leakage policies that are addressed in this chapter are subset of a broader class of policies called information flow control policies. Information flow control policies focus on all possible flow of data among components of a system. These policies even include implicit flow of information [86, 87] or covert channels [88] such as storage channels or timing channels. Unfortunately, covert channels are exceptionally hard to address. In fact, even some of the most sophisticated encryption implementations have been cracked by these implicit channels. Even worse, Schneider et al. [12] has shown that such categories of information flow policies are not enforceable using reference monitor (RM) models and execution monitoring (which we leverage in this dissertation). In designing the intra-service PME framework, we only implemented information flow tracking based on *data-flow dependence*. Explicit data flow from variable x to variable y happens by passing data between fields (variables), or calling methods on such fields. Based on our experience with various web services and SOA implementations, tracking the explicit data flow is sufficient for fine-grained monitoring of almost all use cases. The reason behind it the fact that almost all data leakages in real world scenarios (such as credit card numbers, SSN, etc.), are caused by explicit data flows. However, as a future work, this framework could be extended to support a subset of *control flow dependence* too. Control dependence enables the taint analysis to apply the propagation policies even through predicates (e.g, *if* and *while* blocks) on tainted data.

5.5 Related Work

The concept of information flow control is initially presented by Denning [82] by proposing a formal model for monitoring and enforcement of information flow policies. In this mechanism, we associate (label) every program variable with a class. All of these classes create a DAG called *lattice*. The edges of this lattice specify the way information is allowed to flow inside the program. If there is no violation of the lattice flow, then the program is secure. Even though this paper lays a formal foundation for information flow control, however, it does not provide any practical approach that can be applied to modern web services.

Nair et al. proposed an architecture for information flow control for Java at the bytecode level, called *Trishul* [89]. The main drawback of Trishul is the requirement for changes in the Java runtime environment (JVM). Such a requirement makes the system impractical for real world scenarios.

She et al. describe an information flow control and access policy mechanism called SCIFC for SOA, which is based on delegations and pass-on certificates [90–94]. The goal of this approach is to control the leakage of sensitive data while being transformed by intermediate nodes in a chain of services. The first shortcoming of this approach is the inefficiency. Since it needs to send and validate the certificates in every service of the whole service chain for every request in both forward and backward direction. Another limitation of this approach is the lack of any practical enforcement mechanism. Moreover, the authors assume that the involved services are semi-trusted, which is a strong and unrealistic assumption as trustworthiness of services change dynamically and their status may change from trusted to untrusted in a short period. Finally, they only simulated the proposed approaches and they do not provide any realistic experiments. We have addressed these limitations by a scalable PME framework that leverages a dynamic trust management system.

Information flow control can be implemented at the application level, platform (virtual-machine) level, or OS level. Application-level information flow control makes it specific to the target application and makes it useless for other applications. Another approach is to implement it at the OS level. Two examples of this approach are HiStar [95] and DStar [88], which allow the OS to enforce control flow policies at the OS-level objects (e.g., address spaces, threads, and devices). The main limitation of this approach is the fact that it is very difficult to translate the high-level policies into low-level (OS-level) syscalls. Furthermore, OS-level approaches require a fundamental change in the OS and runtime environment, which requires changes in every piece of software that is going to be supported by them. To address these limitation, we implement the information flow control at the platform level (JVM-level) using AOP that enables us to apply it to all existing codes without any modification in the legacy systems.

There is another category of solutions that try to enforce the information flow control at the programming-language level. The seminal work is presented by Myers [96], which proposes replacement of Denning's coarse-grained clearance levels with generic security labels. In fact they requires changes in the source code and using a specialized compiler to rewrite the program executables. Similar to most of the previous approaches, this requirements makes this approach undesirable for real world SOA scenarios, which impose limitation on access to the source code of the programs or major changes in the execution environments.

5.6 Conclusion

In the previous chapter, we presented the inter-service PME framework, which plays a key role in enabling the clients to enforce their own policies by monitoring the interactions among SOA services. However, to achieve *end-to-end* policy monitoring and enforcement, we should address both the interactions among services (as discussed in Chapter 4) and the flow of information inside individual services. In this chapter, we presented the second part of the PME framework, called intra-service PME framework, which is fine-grained information flow tracking inside applications and services. To demonstrate the effectiveness and performance of this framework, we conducted experiments using the ticket reservation system (as described in the previous chapter) using two different EC2 cloud testbeds (with different computation and memory power).

The experimental study in this chapter showed that our proposed framework is highly efficient and its overhead is lower than 5% under various settings (different testbeds and levels of concurrency). The amount of extra overhead compared to inter-service PME is in the range of $1.7 - 2.9\%$, which is caused by a more complex logic of the taint analysis aspects. The comparison between the results of the two cloud testbed shows that the intra-service PME framework scales linearly (with 85% efficiency) with the number of CPU cores, similar to what we observed for the inter-service PME framework.

As mentioned earlier, the proposed taint analysis mechanism has a very low performance overhead compared to traditional taint analysis frameworks, which usually show down the system $10 - 25$ times and makes them impractical for online deployments and analyses. This improvement has been achieved by implementing the taint analysis at the higher level of abstraction. On the other hand, traditional taint analysis mechanisms operate at the machine-instruction level. In such low-level abstraction, the taint analysis framework is required to instrument *all* CPU instructions blindly, as there is no way to exclude unnecessary parts of the code from the expensive instrumentation. However, we implemented the taint analysis using the AOP, which operates at a higher level of abstraction and which only instruments the necessary parts of the code.

Furthermore, to demonstrate the effectiveness of the intra-service PME framework in preventing data leakage attacks, we developed a leakage attack scenario, in which a service received private information from a client and then manipulates it by applying several operations on it (e.g., cutting a subset of private data) and passing it around in different methods. This web service eventually decides to communicate with another service and tries to send some data, which includes part of the user's private data. At this point, the intra-service PME framework detects that suspicious information disclosure is going to happen, and stops the execution of the target service and consults with the trust manager. Next, the trust manager uses the policy engine to decide whether, according to client's policies, this request is considered malicious or not. Next, in our attack scenario, the trust manager asks the intra-service PME service to prevent this data leakage, which successfully enforces it. We conducted similar scenarios (with different policies and disclosure logic), and the intra-service PME framework was able to detect all of them successfully.

6 ADAPTIVE AND SECURE SERVICE COMPOSITION

In this chapter, we present a policy-based and adaptive secure service composition mechanism for SOA. In this composition approach, system can dynamically change the service topology to achieve a higher composite trust. This dynamic recomposition is triggered by a service consumer through a security policy. The structure of the chapter is as follows. First, we investigate the motivations for designing a policy-based and adaptive secure service composition. Second, We formulate the *secure service composition* as a knapsack problem [97]. Next, we present *EM-HEU* as an efficient heuristic algorithm to maximize the trust in the composite services. Finally, we evaluate the efficiency of the proposed composition approach through a realistic case study.

6.1 Motivation

The main paradigm in SOA is to enable clients to compose services from various service providers to create a larger, more complex application. However, secure service composition in serious scenarios (such as banking and military) is challenging. On the other hand, SOA consumers have different priorities and requirements. For example, one client may prefer a service composition with faster response time, while another client may prefer a composite service with the highest trustworthiness.

The solution to the diversity of requirements by different user is to design an algorithm that captures the users' criteria (and constraints) and then finds an optimal or near-optimal solutions based on the provided user policy. Specifically, in this dissertation, we design a secure service composition mechanism that maximizes the composite trust while satisfies the service consumers other criteria. Such an algorithm leverages a collection of valuable historical data, which is collected by trust manager service from the past execution of services.

Another driving force behind designing an optimal service composition mechanism is the fact that services in SOA are highly dynamic. Therefore, client must be able to reconfigure the service composition on-demand to achieve a higher trust. This property enables a SOA framework to be *adaptive* and react to changes accordingly. For example, once a service is attacked in a SOA scenario, other components of the proposed security framework (such as TM service, policy engine, and PME components) coordinate with each other to detect the policy violations. At this point, client can enables the TM service (by a policy) to invoke the dynamic composition algorithm and calculate the current optimal service composition according to the latest SOA context (QoS and trust parameters of Services) and reconfigure the SOA application accordingly.

However, finding an efficient secure service composition is a challenging problem. The reason is that the number of candidate service compositions grows exponentially with respect to the problem parameters. For example, if we have 10 service categories, each of them have 10 candidate services, and there are 4 constraints (e.g., trust, availability, etc.), then we require to analyze 400 combinations. As we will discuss more formally later in this chapter, the knapsack formulation of the optimal service composition does not have a polynomial-time solution. Therefore, designing an efficient (polynomial-time) heuristic algorithm is highly desired.

6.2 Formulation and Design of Secure Service Composition Algorithm

In service-oriented architecture, every service composition is composed of a series of services that interact with each other based on a service interaction graph. One of the benefits of SOA is the fact that usually, there are multiple selections of services for every required task. We define a *service category* as an abstraction for a set of services that have a similar service access interface and provide a similar functionality. A *concrete service* is a real implementation of that service category.

Since service consumers have different requirements (such as level of service, security assurance, etc.), service providers offer multiple services in the same category. In this business model, depending on the quality of the service and trustworthiness, they charge the service consumers with different rates. Moreover, competing service providers can also deliver similar services. The composition aims at selecting a set of services that provide the highest level of trustworthiness. However, there are some constraints that must be met. One of them is the total affordable cost of service that must not exceed a predefined value C and also QoS constraints.

As we discussed in chapter 2 and 3, trust manager service maintains the latest trust values of services based on their history of execution. It further can maintain the QoS parameters for all services in the orchestration. Therefore, using ASSC, TM service can periodically or based on client's request, find the best composition.

In the simplest form, we just consider the trust value as the only metric. In this case, we replace a service in an orchestration with another service in the same category with the highest level of trust. However, in real world scenarios, the cost of using services put a limit on using an arbitrary service. The problem of *secure service composition* could be formulated as an instance of a *multiple-choice knapsack (MCK)* problem [98] or *multiple-choice multi-dimensional knapsack (MMK)* problem [99]. We will use notations from Table 6.1 throughout this chapter.

Table 6.1: Notations used in ASSC formulation

Notation	Description
S_i	Service category i
s_{i_j}	Concrete service i in category j
t_{ij}	Trust value
c_{ij}	Cost of service i in category j (MCK)
c_{ij}^k	Constraint kth value for service i in category j (MMK)
x_{ij}	1 if service i in category j is selected, otherwise 0
p_{ij}	Profit of choosing service i in category j
w_{ij}	Weight (cost) of choosing service i in category j
m	Number of categories
N_i	Number of services in category i
C	Maximum possible cost of services (MCK)
C^k	Maximum possible value for constraint k for selected services (MMK)

First, we represent the problem as a *0-1 multiple-choice knapsack problem.* The original knapsack formulation is as follows [100]:
There are n items, $1 \leq j \leq n$, each with profit (or value) p_j and weight (or cost) w_j. The decision variable x_j is used to show whether an item is selected ($x_j = 1$) or not ($x_j = 0$). In the MCK formulation, these items are divided into m categories N_i for all $1 \leq i \leq m$. Each item j in category i has a profit p_{ij} and weight w_{ij}. The goal is to select a subset of the items, with highest total profit, while the maximum total weight of the selected items must not exceed W. We assume all coefficients in this formulation are integers (or are converted to integer by proper scaling).

$$maximize \sum_{i=1}^{m} \sum_{j \in N_i} p_{ij} x_{ij}$$

$$subject\ to \sum_{i=1}^{m} \sum_{j \in N_i} w_{ij} x_{ij} \leq W$$

$$\sum_{j=N_i} x_{ij} = 1 \quad for\ all \quad 1 \leq i \leq m$$

$$x_{ij} \in \{0,1\} \ \ for\ all\ 1 \leq i \leq m,\ j \in N_i.$$

MCK Formulation for secure service composition problem. We formulate our service composition problem as followings: *Service Category* is an abstract collection, which represents a set of concrete services with similar functionality. We use $\{S_1, S_2, ..., S_m\}$ to show m categories in the knapsack problem. Concrete services corresponding to the service category S_i are shown by $\{s_{i1}, s_{i2}..., s_{ik}\}$.

$$maximize \sum_{i=1}^{m} \sum_{j \in S_i} t_{ij} x_{ij}$$

$$subject\ to \sum_{i=1}^{m} \sum_{j \in S_i} c_{ij} x_{ij} \leq C$$

$$\sum_{j=N_i} x_{ij} = 1 \quad for\ all \quad 1 \leq i \leq m$$

$$x_{ij} \in \{0,1\} \quad for\ all\ 1 \leq i \leq m,\ j \in N_i.$$

Unfortunately, MCK problem is an NP-complete problem [101]. Therefore, there is no efficient (polynomial-time) algorithm that solves this problem.

The straightforward solution to solve this problem is *dynamic programming* technique, which gives a *pseudo-polynomial* solution [101, 102]. The formulation of our problem in dynamic programming is as followings:

In this formulation, k is the number of selected categories ($1 \leq k \leq m$) and y is the total cost of selected services ($0 \leq y \leq C$).

We consider the optimal value of the formulated problem for k and y be $v_k(y)$. Then, the dynamic programming recursive formula is as following:

$$v_k(y) = \begin{cases} v_0(y) = 0, & \text{if } y \geq 0 \\ v_k(y) = -\infty, & \text{if } y \leq 0\ and\ i > 0 \\ \max\limits_{j \in N_k} \{t_{kj} + v_{k-1}(y - c_{kj}) \leq n\}, & \text{otherwise} \end{cases}$$

In this formula, $v_m(C)$ gives the optimal value of the trustworthiness. The complexity of this solution is $O(nC)$, which n is the total number of services in the system and C is the total affordable cost. This solution is pseudo-polynomial since if C is known to not grow faster than a polynomial function of n then its complexity is polynomial. However, the complexity in general is exponential. However, There are some approximation algorithms that solve this problem in polynomial time. For example, authors in [101] presented a branch and bound algorithm with complexity of $O(mlog^2(n/m))$. Furthermore, in the same paper, an approximation algorithm is proposed that solves this problem in linear time by a linear programming relaxation technique.

The main limitation of this knapsack formulation is that it only considers a single constraint. To relax this assumption, we reformulate the problem as a multidimensional multiple-choice knapsack (MMK) problem. The formulation is as follows:

In addition to *cost*, as a selection constraint, we are interested in considering other QoS constraints in the optimal secure service composition problem. QoS constraints are generally available in the contracts between service publishers and service consumers, which sometimes are called SLAs (service-level agreements). Since there is no guarantee that a service provider follows its SLA, we collect these QoS parameters by PME framework, which are based on actual execution of services, and maintain them in the TM service.

In this dissertation, we are interested in the following SLA (QoS) parameters (but, the list could be extended without affecting on the operation of the algorithm):

Cost This parameter represents the monetary cost of a service. We represent the cost of service S_i by C_i, which is the cost of using service per invocation. Such monetary schemes could be more complex in the real-world scenarios and may depend on different level of SLA. However, to simplify the presentation of the proposed composition mechanism, we assign a constant cost C_i to a service S_i.

Delay (D) This parameter represents the response time of a service.

Note: The *cost* and *delay* parameters are additive. It means the cumulative parameter for n services is calculated by summing up the individual parameters such as $\sum_{i=1}^{n} c_i$ gives the total cost. However, some of the SLA parameters are not additive. For example, service reliability is multiplicative ($R_{total} = \prod_{i=1}^{n} R_i$). Therefore, we can use logarithmic values of multiplicative parameters (e.g. using $log(R)$ instead of R) to treat all SLA parameters similarly as additive in the proposed algorithms.

MMK Formulation for secure service composition problem. The formulation for this problem is similar to the previous problem, except that there are p formulas to represent p constraints. But, to represent the problem more concisely, we use a superscript variable. Similarly, we assume there are m service categories, $\{S_1, S_2, ..., S_m\}$, in the knapsack problem. Concrete services are similarly represented by $\{s_{i1}, s_{i2}..., s_{ik}\}$ for service category S_i.

$$maximize \sum_{i=1}^{m} \sum_{j \in S_i} t_{ij} x_{ij}$$

$$subject\ to \sum_{i=1}^{m} \sum_{j \in S_i} c_{ij}^{k} x_{ij} \leq C^k \quad for\ all \quad 1 \leq k \leq p$$

$$\sum_{j=N_i} x_{ij} = 1 \quad for\ all \quad 1 \leq i \leq m$$

$$x_{ij} \in \{0,1\} \;\; for\ all\ 1 \leq i \leq m,\ j \in N_i.$$

MMK problem is also a NP-hard problem [99]. Authors in [103] proposed a *dynamic programming* technique for this problem. Several heuristics are also presented to solve the approximations of this problem efficiently [104–106].

We use *M-HEU* [105] that is a very efficient heuristic algorithm to solve the MMK problem. M-HEU algorithm is an optimized variation of *HEU* algorithm [107]. To gain a faster (and time-bounded) solution, we modified the M-HEU by limiting the number of upgrades and the number of downgrades at the expense of losing a negligible optimality of a solution.

We improve the *M-HEU* by adding a preprocessing step and also improving the selection of the initial feasible solution. In the following formulation, p specifies resources, $\Delta_{ij}a$ shows an aggregate weight exhaustion, and finally $\Delta t_{ij} = t_{i\mu_i} - t_{ij}$ is the amount of gain in the total trustworthiness.

$$\Delta a_{ij} = \frac{\sum\limits_{k=1}^{p} (c^k_{i\rho[i]} - c^k_{ij}) C^k}{|C|}$$

Where $|C'| = \sqrt{(C'^1)^2 + (C'^2)^2 + ... + (C'^k)^2}$ is the current exhausted QoS bounds.

The runtime complexity of the EM-HEU algorithm is $pm^2(L_{max} - 1)^2$ where $L_{max} = \max\limits_{1 \leq i \leq m} |S_i|$. There is a factor of $O(pn^2)$ for the preprocessing, which is dominated by a larger (or equally large) factor. We omit the detailed derivation of the complexity since it is similar to the derivation in [105].

Akbar et al. [105] proposed an incremental heuristic algorithm to solve the multidimensional knapsack problem. This algorithm is particularly interesting since in our use case the trust values change frequently.

6.3 Implementation and Evaluation

6.3.1 Implementation

The EM-HEU algorithm is implemented as a generic algorithm, which can operate on an arbitrary number of services and QoS constraints. We implemented the ASSC as a web service inside the TM service, which is accessible through four REST APIs:

- `POST /setServiceTrusts` This API accepts a map of services and their trust values. The trust values could be externally set by clients or it could be used directly inside the TM service.
- `POST /setOptions` This API is used to set the number of QoS constraints and the corresponding values for each service.

Algorithm 3 Part 1– EM-HEU Algorithm Enhanced greedy Algorithm for MMKP

```
1: **Input:** A set of $m$ service categories $S_i$, each has a set of services.
2: **Output:** A set of services, one in each category, with near-optimal trustworthiness.
3: **Step 1:**        ▷ Preprocessing inputs.
4: preprocess();
5: **Step 2:**        ▷ Finding a feasible solution.
6: **step 2.1:**        ▷ Fast and randomized search for a feasible solution.
7: Selecting a service from each category randomly. The index of the selected concrete service in category $i$ is $\mu_i$.
8: **step 2.2**        ▷
9: **if** $(\sum_{i=1}^{m} c^k_{i\mu_i} \leq C^k)$ **then**        ▷ for all $1 \leq k \leq p$
10:    print(”This problem has a solution!”);
11:    goto *step 3*;        ▷ selected solution is feasible, then skip the rest of step 3.
12: **end if**
13: **step 2.3**        ▷
14: Selecting the lowest-value services (highest-cost) from each category $1 \leq i \leq m$: $\max_{1\leq k\leq p}\{f_k\}$, which $f_k = c^k_{ij}/C^k$ is infeasibility factor.
15: **if** all possible services are already selected **then**
16:    print(”This problem has no solution.”);
17:    exit();
18: **else**
19:    goto step 2.2;
20: **end if**
21: **Step 3**        ▷ Improving the selected solution by upgrade process
22: **step 3.1** Finding a higher value service from a category that the selected service of that category is subject to the QoS constraints (step 2.2) and has the highest $\Delta a_{ij}$.
23: **if** there is no such service **then**
24:    select a service with the highest $\Delta t_{ij}/\Delta a_{ij}$.
25:
26: **end if**
```

Algorithm 4 Part 2– EM-HEU Algorithm Enhanced greedy Algorithm for MMKP

```
27: if no service is found in step 3.1 then
28:     goto step 4;
29: else
30:     Find another service in step 3.1;
31: end if
32: Step 4        ▷ Improving the solution by one upgrade succeeded by at least one downgrade
33: step 4.1 Finding the highest value services among the selected services in any category, which has the highest $\Delta t_{ij}/\Delta a' c_{ij}$
34: $\Delta a'_{ij} = \sum_{k=1}^{p} \frac{(c^k_{i\mu_i} - c^k_{ij})C^k}{C^k - C'^k}$
35: step 4.2 Finding a lower-valued service in any service category with the highest value that keeps the downgrade still have a greater value than the second step and maximizes the $\Delta a''_{ij}/\Delta t_{ij}$ $\Delta a''_{ij} = \sum_{k=1}^{p} \frac{(c^k_{i\mu_i} - c^k_{ij})C^k}{C'^k - C^k}$
36: if the output of 4.2 $\neq \varnothing$ then
37:     if the service fulfills the QoS constraints then
38:         go to step 3 to search for a more optimal solution
39:     else
40:         go to 4.2 for further downgrade.
41:     end if
42: else
43:     Print(solution found at the end of step 3);
44:     exit();
45: end if
```

Algorithm 5 Preprocessing the input for EM-HEU algorithm.

```
1:  **Input:** A set of $n$ concrete services ($s_1 - s_n \in S$) and $p$ SLA constraints($C_1 - C_p$)
2:  **Output:** A set of services $S$, with all non-compliant services removed.
3:  **procedure** preprocess()
4:  **Step 1**      ▷ First removing services that explicitly violate SLA parameters
5:      **for** $i \leftarrow 1$ *to* $n$ **do**      ▷ $n$ is the total number of services
6:          **for** $j \leftarrow 1$ *to* $p$ **do**      ▷ $p$ is the number of SLA constraints
7:              **if** $s_i.SLA_j > C_j$ **then**
8:                  $S \leftarrow S - s_i$
9:              **end if**
10:         **end for**
11:     **end for**
12: **Step 2**      ▷ Next, removing services that are *dominated* by other services.
13:     **for** $i \leftarrow 1$ *to* $n$ **do**      ▷ $n$ is the total number of services
14:         **for** $j \leftarrow i$ *to* $n$ **do**
15:             **if** $s_i.SLA_k > s_j.SLA_k$ does not hold for all $1 \leq k \leq p$ **then**
16:                 $S \leftarrow S - s_i$
17:             **end if**
18:         **end for**
19:     **end for**
20: **Step 3**      ▷ Finally sorting the remaining services in $S$ for faster operation of
    *EM-HEU* algorithm. Sorting will be based on all SLA parameters (producing $p$
    sorted list)
21:     sort($S$)
22: **end procedure**
```

- `POST /setResources` This API is used to set a limit for the total resources that the selected services must comply.
- `GET /getASSC` This API is called after calling the previous APIs and it gives an optimal service composition among the available services while it fulfills the constraints. If there is no such a solution, it returns null.

In the UI demo, we selected two constraints. The first parameter is the execution time of services, the second parameter is the cost of each service, which will be provided by clients. Collecting the service execution time is implemented through an aspect. Figure 6.1 shows a pointcut definition, which is used to instrument the Jersey REST framework and collect the execution time of the POST calls. Once this pointcut is matched, the `execTimeMeasure()` advice is called. Figure 6.2 gives a specific aspect to calculate and store the response time of the services. After calculating the response time, it will be reported to the TMService, which maintains the QoS measurements.

```
<aop>
   <pointcut name=ServiceCall" expr="call(* com.sun.jersey.api.client.WebResource.
    Builder->post(..))"/>
    <bind pointcut="ServiceCall">
        <around name="execTimeMeasure" aspect="edu.purdue.cs.soa.pme.QoSAspect />
    </bind>
</aop>
```

Figure 6.1.: Execution time monitoring using AOP

```
//pointcut: @around("call(* com.sun.jersey.api.client.WebResource.Builder->post
    (..)))
public Object execTimeMeasure(CallerInvocation invocation) throws Throwable {
  ...
  long start_time = System.currentTimeMillis();
  Object ret = invocation.invokeNext(); //executing the service invocation method
  long end_time = System.currentTimeMillis();
  reportTMService("runtime", (end_time - start_time), context);
  return ret;
}
```

Figure 6.2.: Measuring the execution time of an operation using AOP

6.3.2 Performance Evaluation

In this section, we discuss the performance evaluation of the ASSC component.

Goal of the experiment. The goal of this experiment is to verify the efficiency of the ASSC component in term of response time.

Method of the experiment. To evaluate the performance of the ASSC component, we perform the performance evaluation of the ASSC component in the EC2 cloud. Similar to the experiments in the previous chapters, we deployed the policy engine and TM service in a t2.small EC2 instance. The other parameters are presented in Table 4.4. To minimize the effect of unpredictable and variable delays on the client side, the client (Apache Benchmark [62]) is also deployed in a separate t2.micro virtual machine. We have repeated each experiment 5000 times and the results are the average metrics of these experimental runs.

Input parameters of the experiment. There are three input parameters that we will investigate their effects on the performance of the ASSC component. These parameters are: total number of services, number of categories, and number of constraints.

Results, analysis, and conclusions of the experiment. In this section, we present empirical results of the performance evaluation of the ASSC component based on the previous input parameters.

In the first experiment, we investigate the effect of the number of services on the performance of ASSC. In this experiment, we have set the number of categories to 5 and the number of constraints to 3. Figure 6.3 shows the response time of the ASSC REST API for scenarios with total number of services from 25 to 125. The results show that the execution time changes almost linearly. Even for 125 services in 5 categories (which is unlikely to be surpassed in any practical SOA scenario), the ASSC service performs very well and the average response time is 22ms.

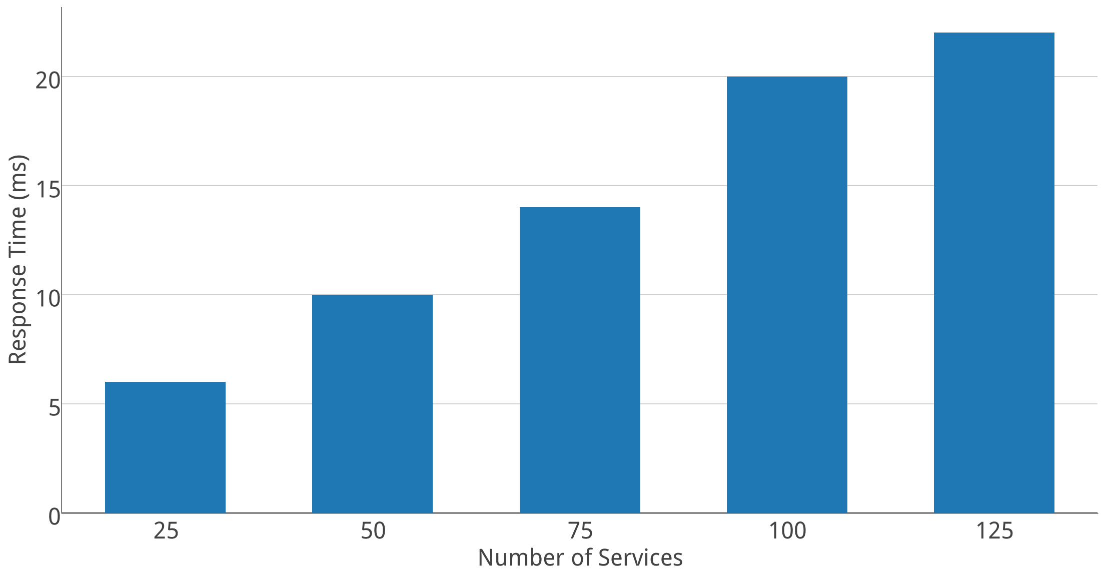


Figure 6.3.: Response-time of the ASSC component in the Amazon EC2 testbed 1 with 5 service categories and 3 constraints.

In the second experiment, we investigate the effect of the number of service constraints on the performance of ASSC component. In this experiment, we have set the number of services to 50 and the number of categories to 5. According to Figure 6.4, the effect of the QoS constraints on the performance of ASSC component is sublinear. Even after increasing the input size by a factor of 5, the response time only increases 50%.

In the third experiment, we investigate the effect of the number of service categories on the performance of ASSC component. In this experiment, the number of

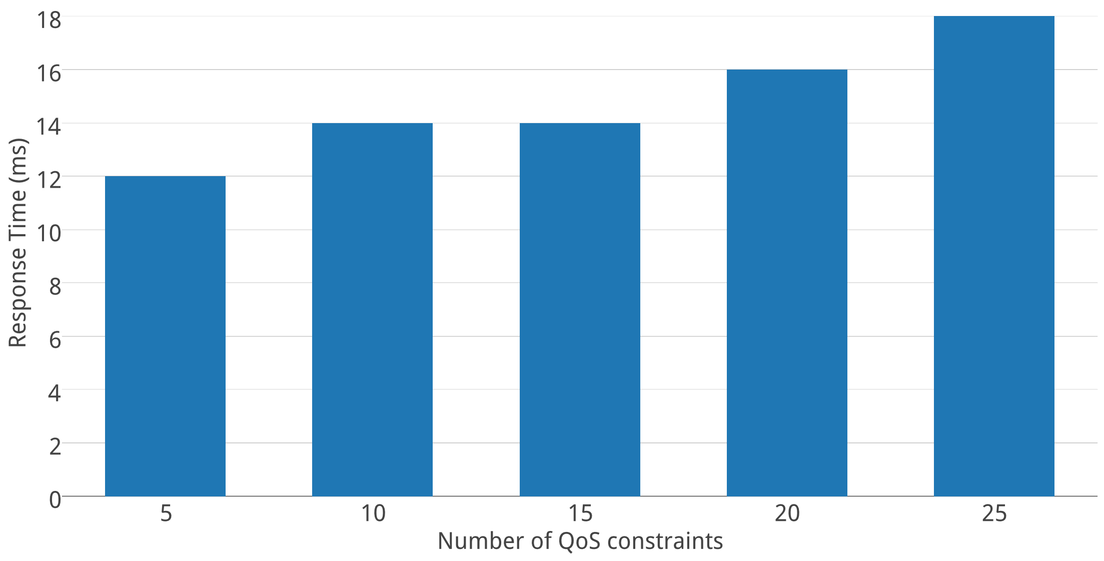


Figure 6.4.: Response-time of the ASSC component in the Amazon EC2 testbed 1 with 50 services and 5 service categories.

services is set to 50 and set the number of constraints is set to 3. Similar to the previous results, based on Figure 6.5, the ASSC response times is fast and it does not exceed 20ms for 25 categories.

All previous performance evaluation experiments confirm that the ASSC component is very responsive for any practical SOA application.

6.3.3 Security Evaluation

Goal of the experiment. In this experiment, we evaluate the effectiveness of the ASSC component in improving the security of the SOA. We expect to see the ASSC component generates a new service compositions (by changing one or more services in the current service orchestration) according to the context (input parameters) to improve the trustworthiness of the SOA. We further want to study the quantitative improvement in the composite trust of the SOA application after reconfiguration.

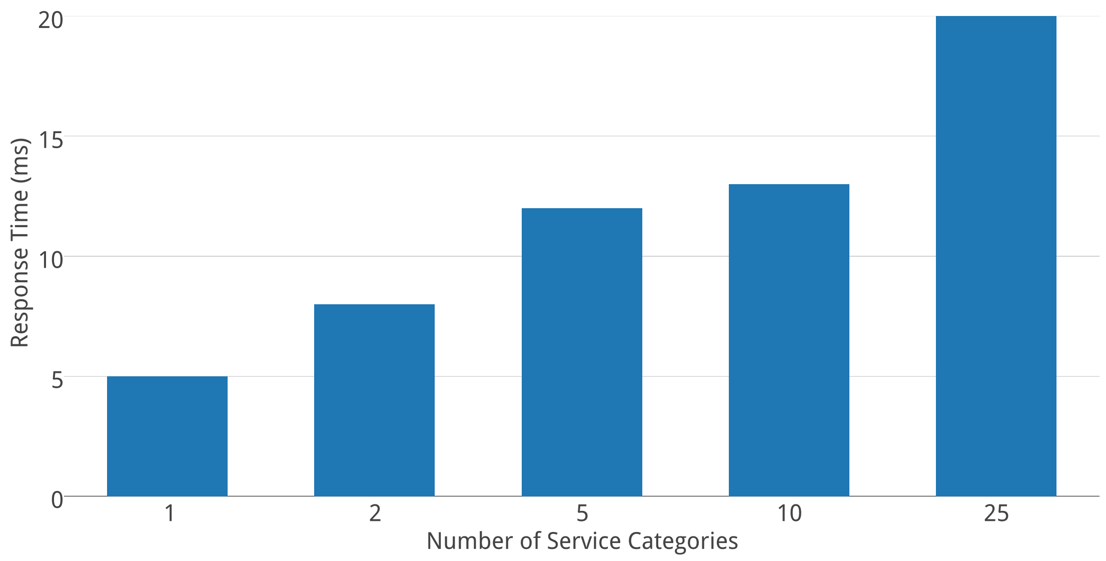


Figure 6.5.: Response-time of the ASSC component in the Amazon EC2 testbed 1 with 50 services and 3 service constraints.

Method of the experiment. For this experiment, we deploy the ticket reservation case study (as described in chapter 4 Figure 4.5). In this experiment we will change the input parameters and we study the generated optimal service composition by ASSC component. We also have implemented a DoS attack, which could be applied on any of the available services.

Input parameters of the experiment.

There are four sources of input for this experiment as follows:

- trust values of all available services.
- execution times (delay) of each available service.
- the cost of each service
- the total acceptable delay and cost for the new service composition.

Results, analysis, and conclusions of the experiment. In addition to the performance evaluation of the experiments which were discussed earlier, we also evaluated the security measures provided by the PME framework on multiple service graphs. In this section, we briefly discuss them using the ticket reservation case study presented in Figure 4.5.

As we discussed in chapter 4, we presented two corrective security measures in this dissertation. ASSC is the second corrective measure, which tries to improve the security by reconfiguring the SOA instead of only blocking the malicious behaviors. We conducted a large number of experiments by changing different subsets of inputs. These experiments are conducted on both ticket reservation case study and also generic DAG scenarios. All experiments confirmed the effectiveness of the ASSC in reconfiguring the SOA in reaction to the changed context. Here, we present one of the scenarios. Figure 6.6 shows an instance of the ticket reservation case study, which does not take advantages of best available services. Figure 6.7 shows a screenshot of the session report after calling the ASSC framework. This figure shows that the composite trust has improved from 0.55 to 0.86 after reconfiguration. In this experiment, we have set the total cost and total service delay parameters large enough to make all services eligible for selection.

The experiments presented in this section, confirm that the PME framework is practical and effective, and scalable.

6.4 Alternative Formulation

In the previous section we presented a formulation, which maximizes the average trust in a service composition. However, as we discussed in chapter 2 (Composite trust strategies), we may decide to use other composite trust strategies such as weighted averaging or pessimistic. In this section, we briefly discuss how we can formulate the ASSC problem using these strategies.

Weighted averaging strategy. Services have different importance in a composition graph of a SOA application. To take this difference into account, as we discussed in chapter 2, we can leverage *pagerank algorithm* [43] to normalize and weight the trust values of services. In the following preprocessing step we assume that τ_{policy} is

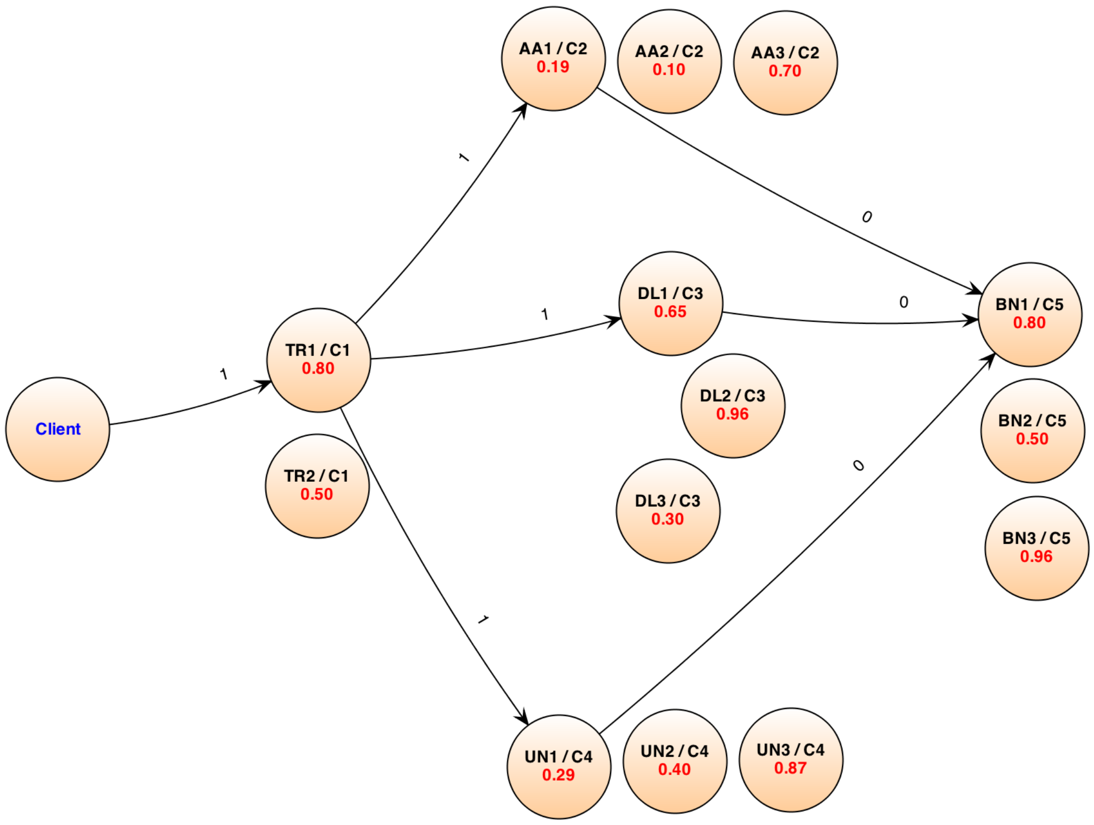
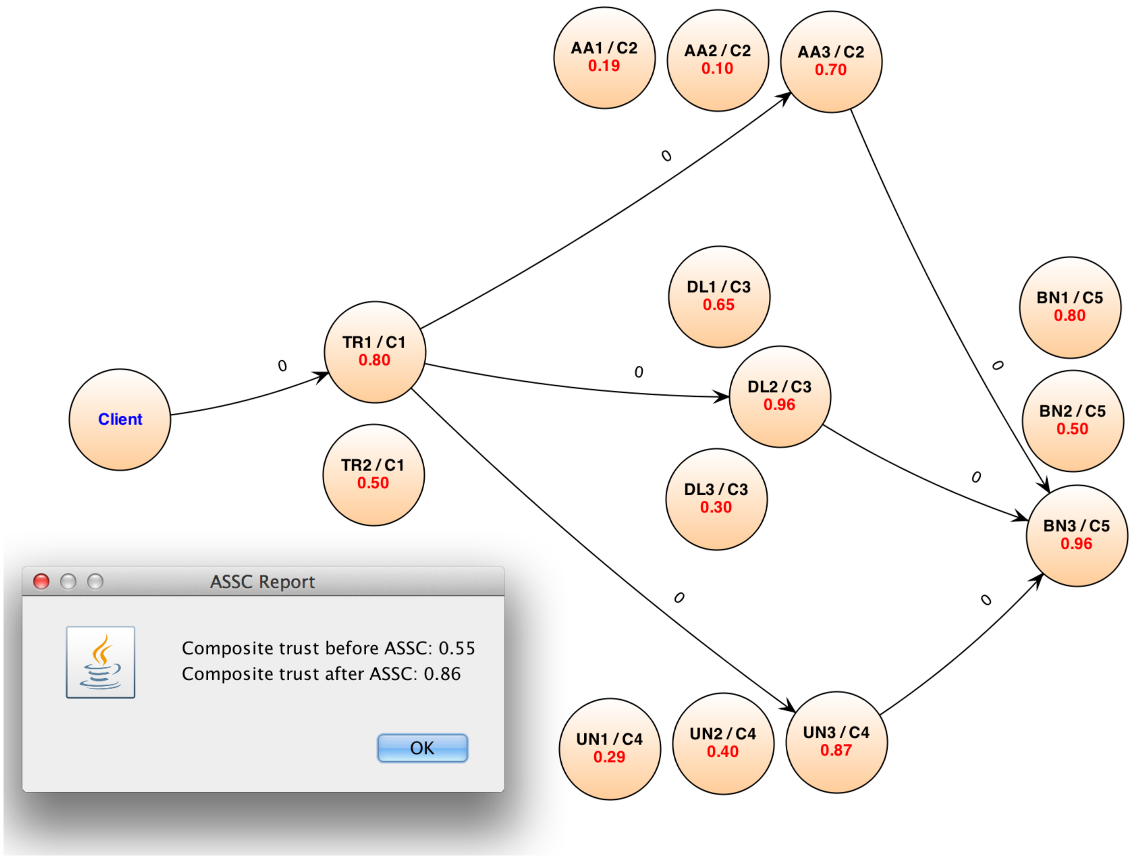


Figure 6.6.: Screenshot of the ticket reservation case study before applying the ASSC.

the trust threshold, $w_i^{'} = \tau_{policy}/t_i$, $r_1, r_2, ..., r_n$ $(\sum_{i=1}^{n} r_i = 100)$ are the PageRanks of services, and $w_i = r_i/10$. In chapter 2, we proposed $(1/w_i^{'})^{w_i}$ as a representative weight for the averaging strategy. [1]

This procedure can be implemented as an extra step in the ASSC preprocessing algorithm.

[1]In chapter 2, we discuss these parameters and the reasoning behind such weighting mechanism.



Composite trust before ASSC: 0.55
Composite trust after ASSC: 0.86

Figure 6.7.: Screenshot of the ticket reservation case study after applying the ASSC.

Algorithm 6 Suggested preprocessing step for weighted averaging strategy.

```
1: Input:
2: $(t_{s_1}...t_{s_n})$: The list of $n$ trust values for concrete services in a service composition
3: $(r_1, r_2, ..., r_n)$: PageRanks of the $(s_1...s_n)$
4: $\tau_{policy}$: Trust threshold
5: Output:
6: $(t'_{s_1}...t'_{s_n})$A list of weighted trust values for the $(s_1...s_n)$
7: Weighting Step        ▷ This step can be used as the first step of Algorithm 5
8: $w_i \leftarrow r_i/10$
9: $w'_i \leftarrow \tau_{policy}/t_i$
10:     for $i \leftarrow 1$ to $n$ do
11:         $t'_i \leftarrow (1/w'_i)^{w_i} \times t_i$
12:     end for
13: return $(t'_{s_1}...t'_{s_n})$
```

6.5 Related Work

Service composition in web services and SOA is a challenging research area that has attracted many researchers in the recent years. Dustdar et al. provides a comprehensive survey of web service composition algorithms in [108]. They categorize the techniques into static, dynamic, declarative, and model-driven. In the static service composition, service selection happens once at the deployment time and remains fixed during the lifetime of its execution. Microsoft Biztalk server [109] is an example of static service composition. However, dynamic service composition assumes services change dynamically and frequently. Therefore, service composition must react dynamically to these changes. Even though the proposed ASSC falls in the dynamic category, however, none of the discussed mechanisms in the survey addresses the problem of optimal service composition.

There are a number of research work that try to formulate the service composition problem as an optimization problem [110–113]. Authors in [110, 111] present two similar formulations of multiple knapsack problem for web service composition. However, they only discuss the problem in the context of QoS and do not address the security criteria. Moreover, their formulations are based on resource requirements of the services, which are not easily obtainable in real-world scenarios at runtime. Finally, none of them has evaluated the proposed algorithms to a real world case study. On the other hand, the proposed ASSC approach is based on the SLA measurements that are maintained by the trust manager. Authors in [114, 115] provide a survey of QoS-based service compositions. Cardellini et al. [113] presented a framework for runtime QoS-driven service composition and adaptation in SOA, which uses linear programming, and solve it using the standard tools.

From the theoretical perspective, authors in [99,116,117] provide efficient heuristic approaches to solve the MMK problem that can be used to reduce the overhead of service selection at the expense of achieving a near-optimal solutions. A number of other optimization approaches has been used to solve the problem of service composition.

For example, [113] uses linear programming, [118] uses Expectation Maximization (or EM) algorithm.

6.6 Conclusion

Selecting an optimal service composition in SOA while meeting the QoS and cost constraints is a challenging problem. In this chapter, we presented the *ASSC engine*, which leverages the trust manager to collect the information about services and employs an efficient heuristic algorithm to solve the secure service selection problem.

To measure the performance of the ASSC engine, we have developed an AOP-based monitoring aspect that is included in the PME modules. This aspect, measures the response time of external service invocations that happen within a service and reports them back back to the trust manager. The trust manager keeps track of the latest response time for every service. In this experiment, we have used the first Amazon EC2 testbed (as described in chapter 4). Each experiment is repeated 5000 times and the final response time is the average of these response times. The ASSC engine has three main independent parameters: the total number of services, the number of service categories, and the number of QoS constraints. The response time of the ASSC engine increases almost linearly according to the number of services. In one experiment, once we increase the number of services from 25 to 125, the response time increases from 6ms to 22ms. The ASSC is much less sensitive to the number of QoS parameters. In the second experiment, we have increased the number of QoS constraints from 5 to 25, but the response time only increases 50%. Finally, once we increase the number of service categories from 1 to 25, the response time of the ASSC engine increases from 5ms to 20ms. All of our measurements show that the overhead is consistently under 25ms for any practical scenario.

In addition to the performance measurement, we evaluated the effectiveness of the ASSC component in improving the overall trustworthiness of SOA applications. We conducted our experiments in the first Amazon EC2 testbed (as described earlier). In this experiment, at each round, we have changed the trustworthiness and QoS parameters of one or more services randomly to see how ASSC reacts. In this experiment, whenever a better service composition was available, the ASSC was able to find it and reconfigure the SOA topology accordingly. Moreover, we studied the reaction of the ASSC system in response to DoS attacks. The DoS attack targets a subset of services and slows them down. ASSC was able to react correctly to these attacks and reconfigure the SOA to avoid non-responsive services. Eventually, once the attack stops, the ASSC engine is able to revert the configuration to the previous optimal state.

7 CONCLUSION AND FUTURE WORK

7.1 Contributions

This dissertation aimed to address the challenges concerning end-to-end security in SOA. In summary, it presents the following contributions to the research on SOA security:

- In chapter 1 and 2, we thoroughly studied the problem of end-to-end security in SOA and investigated the corresponding challenges, security requirements and design principles in designing a practical security frameworks. We further designed a policy-based architecture that meets the investigated requirements and design principles. This security architecture is composed of a trust management framework, two policy monitoring frameworks (inter-service and intra-service), a policy enforcement framework, and finally a XACML policy engine. Throughout this dissertation, we referenced the design principles as a guideline in designing the relevant security frameworks.

- In chapter 3, we introduced a new dynamic trust management framework (called trust manager) for trust evaluation and maintenance in SOA. This component plays a key role in maintaining end-to-end security in SOA. As a part of this framework, we designed and implemented session management subsystem and trust maintenance subsystem. The first component, session management subsystem, maintains a chain of trust for tracking a SOA invocation end-to-end. The second component, trust maintenance subsystem, updates the trust based on a trust scheme. We further propose several composite trust schemes (strategies and algorithms) based on graph abstraction.

- In chapter 4, we proposed an inter-service policy monitoring and enforcement framework for SOA, which is able to monitor and enforce the client's policies at runtime. In the inter-service monitoring mechanism, the PME components inspect the external service invocations in all participating services and report the interactions to trust manager. We extended the monitoring framework to be able to enforce security policies in SOA. Moreover, we investigated the enforceable policies by AOP and presented various enforcement strategies. Finally, we implemented, evaluated the proposed enforcement framework to validate its efficiency and effectiveness.

- In chapter 5, an intra-service monitoring and enforcement framework is proposed. This framework tracks the flow of client's sensitive information inside a service and once that service is going to be disclosed to an external service, the PME component reports the incident to trust manager and blocks the leakage. In this chapter, we have studied the effectiveness and practicality of the proposed framework through a realistic case study.

- Finally, in chapter 6, we focus on designing an adaptive and secure service composition engine, which is able to find optimal service compositions with maximum composite trust. This problem of secure service composition is formulated as generalized knapsack optimization problems and an efficient heuristic algorithm is presented.

7.2 Future Work

There are a number of ways to enhance the contributions of this dissertation. The list of tasks is as follows:

1. *Designing advanced anomaly detection engine.* In chapter 4 of this dissertation, we discussed a generic policy monitoring framework, which collects the security events and send them to trust manager. Over time, trust manager will

have a valuable collection of security events, which can be analyzed to further detects attacks and other anomalies in the system. For this purpose, a rule engine (e.g., JBoss Drools) or a complex event processing engine (e.g., Esper) could be utilized. This event analyzer system could be integrated with a security information and event management (SIEM) [119] system, or an enterprise monitoring system (e.g., ganglia [120]).

Testing and fault injection in SOA. The proposed taint analysis framework can be used for service-level testing, similar to Netflix Chaos Monkey [121] for the cloud infrastructure instances. Services can be terminated on-demand or randomly to analyze the effect of the potential failure in the system. This feature could be implemented by performing fault injection [122] (e.g., termination of service, exception, etc.) at the policy enforcement points (PEPs), which intercepts the execution of services.

2. *Extending the framework to support other languages.* As we mentioned in **??**, AOP is widely available in almost all major programming languages and platforms. One interesting next step would be to extend the current infrastructure to support other platforms, such as .NET framework, C/C++, Ruby, and Python.

LIST OF REFERENCES

LIST OF REFERENCES

[1] SOA market share. `http://www.zdnet.com/blog/service-oriented/soa-market-growing-17-a-year-to-reach-10-billion-by-2015/1931`. Accessed March 2014.

[2] David Chappell. *Enterprise Service Bus*. O'Reilly Media Inc., 2004.

[3] Stephen White. *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Future Strategies Inc., 2008.

[4] Trusted Computing Group (TCG). `http://www.trustedcomputinggroup.org/`. Accessed March 2014.

[5] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. Principles of remote attestation. *International Journal of Information Security*, 10(2):63–81, 2011.

[6] Agreiter Berthold, Muhammad Alam, Ruth Breu, Michael Hafner, Alexander Pretschner, Jean-Pierre Seifert, and Xinwen Zhang. A technical architecture for enforcing usage control requirements in service-oriented architectures. In *Proceedings of the 2007 ACM Workshop on Secure Web Services*, pages 18–25, 2007.

[7] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation: A virtual machine directed approach to trusted computing. In *USENIX Virtual Machine Research and Technology Symposium*, 2004.

[8] John Lyle and Andrew Martin. On the feasibility of remote attestation for web services. In *International Conference on Computational Science and Engineering, (CSE'09)*, volume 3, pages 283–288, 2009.

[9] Bryan Parno, Jonathan M McCune, and Adrian Perrig. Bootstrapping trust in commodity computers. In *IEEE Symposium on Security and Privacy (SP)*, pages 414–429, 2010.

[10] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. vTPM: Virtualizing the trusted platform module. In *Proceedings of 15th USENIX Security Symposium*, pages 305–320, 2006.

[11] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2012.

[12] Fred B Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[13] E Rissanen. Extensible access control markup language (XACML) version 3.0. Technical report, OASIS `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-os-en.html`, 2012. Accessed March 2014.

[14] SunXACML: an open source XACML policy engine. `http://sunxacml.sourceforge.net/`. Accessed March 2014.

[15] Balana: an open source XACML 3.0 policy engine. `https://github.com/wso2/commons/tree/master/balana`. Accessed March 2014.

[16] WSO2 Identity Server. `http://wso2.com/products/identity-server/`. Accessed March 2014.

[17] Health Insurance Portability and Accountability Act of 1996 Public Law 104-191. `http://www.gpo.gov/fdsys/pkg/PLAW-104publ191/html/PLAW-104publ191.htm`. Accessed March 2014.

[18] Mehdi Azarmi, Bharat Bhargava, Pelin Angin, Rohit Ranchal, Norman Ahmed, Asher Sinclair, Mark Linderman, and Lotfi Ben Othmane. An end-to-end security auditing approach for service oriented architectures. In *IEEE 31st Symposium on Reliable Distributed Systems (SRDS)*, pages 279–284, 2012.

[19] Gabriela Gheorghe. *Security policy enforcement in service-oriented middleware*. PhD thesis, University of Trento, Italy, 2011.

[20] Push technology. `http://en.wikipedia.org/wiki/Push_technology`. Accessed September 2014.

[21] Peter Mell and Tim Grance. The NIST definition of cloud computing. *Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology*, 2011.

[22] Jelena Mirkovic, Sven Dietrich, David Dittrich, and Peter Reiher. *Internet Denial of Service: Attack and Defense Mechanisms*. Prentice Hall, 2004.

[23] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert Van Doorn. Attestation-based policy enforcement for remote access. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 308–317, 2004.

[24] S Wang, Zibin Zheng, Zhengping Wu, M Lyu, et al. Reputation measurement and malicious feedback rating prevention in web service recommendation systems. *IEEE Transactions on Services Computing*, 2014.

[25] Web Services Security: SOAP message security 1.1 (WS-Security 2004) OASIS standard specification. `http://www.oasis-open.org/committees/download.php/16790/wss-v1`. Accessed July 2014.

[26] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, and Hans Granqvist. WS-Trust 1.3. *OASIS Standard*, 19, 2007.

[27] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, and Hans Granqvist. WS-SecureConversation 1.3. *OASIS Standard*, 1, 2007.

[28] K Lawrence, C Kaler, A Nadalin, M Goodner, M Gudgin, A Barbir, and H Granqvist. WS-SecurityPolicy 1.3. *OASIS Standard*, pages 41–44, 2009.

[29] Azzedine Benameur, Faisal Abdul Kadir, and Serge Fenet. XML rewriting attacks: Existing solutions and their limitations. *In IADIS Applied Computing*, 2008.

[30] Smriti Kumar Sinha and Azzedine Benameur. A formal solution to rewriting attacks on SOAP messages. In *Proceedings of the 2008 ACM Workshop on Secure Web Services*, pages 53–60, 2008.

[31] Matthew Burnside and Angelos D Keromytis. F3ildcrypt: End-to-end protection of sensitive information in web services. In *Information Security*, pages 491–506. Springer, 2009.

[32] Christian Schneider, Frederic Stumpf, and Claudia Eckert. Enhancing control of service compositions in service-oriented architectures. In *International Conference on Availability, Reliability and Security, (ARES'09)*, pages 953–959, 2009.

[33] Cisco Enterprise Policy Manager. `http://www.cisco.com/c/en/us/products/security/index.html`. Accessed March 2014.

[34] Cisco ISE (Identity Services Engine). `http://www.cisco.com/c/en/us/products/security/identity-services-engine/index.html`. Accessed March 2014.

[35] IBM Tivoli Security Policy Manager. `http://www-03.ibm.com/software/products/en/security-policy-manager/`. Accessed March 2014.

[36] Wattana Viriyasitavat and Andrew Martin. A survey of trust in workflows and relevant contexts. *IEEE Communications Surveys & Tutorials*, 14(3):911–940, 2012.

[37] Rafae Bhatti, Elisa Bertino, and Arif Ghafoor. A trust-based context-aware access control model for web-services. *Distributed and Parallel Databases*, 18(1):83–105, 2005.

[38] Claudio A Ardagna, Sabrina De Capitani di Vimercati, Stefano Paraboschi, Eros Pedrini, Pierangela Samarati, and Mario Verdicchio. Expressive and deployable access control in open web service applications. *IEEE Transactions on Services Computing*, 4(2):96–109, 2011.

[39] Hai-bo Shen and Fan Hong. An attribute-based access control model for web services. In *7th International Conference on Parallel and Distributed Computing, Applications and Technologies, (PDCAT'06)*, pages 74–79, 2006.

[40] Richard S. Bird. Notes on recursion elimination. *Communications of the ACM*, 20(6):434–439, 1977.

[41] CW Flink. Weakest link in information system security. In *Workshop for Application of Engineering Principles to System Security Design (WAEPSSD)*, 2002.

[42] Djamal Benslimane, Schahram Dustdar, and Amit Sheth. Services mashups. *The New Generation of Web Applications*, pages 13–15, 2008.

[43] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. *Stanford InfoLab*, 1999.

[44] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Stein. *Introduction To Algorithms*. MIT Press, Cambridge, 3rd. edition, 2009.

[45] Christian Borgs, Michael Brautbar, Jennifer Chayes, and Shang-Hua Teng. A sublinear time algorithm for pagerank computations. In *Algorithms and Models for the Web Graph*, pages 41–53. Springer, 2012.

[46] Jersey: A RESTful web service framework for Java. `https://jersey.java.net/`. Accessed July 2014.

[47] Moving average. `http://en.wikipedia.org/wiki/Moving_average`. Accessed March 2014.

[48] Zaki Malik and Athman Bouguettaya. RATEWeb: Reputation assessment for trust establishment among web services. *The International Journal on Very Large Data Bases*, 18(4):885–911, 2009.

[49] George Spanoudakis and Stephane LoPresti. Web service trust: Towards a dynamic assessment framework. In *International Conference on Availability, Reliability and Security, (ARES'09)*, pages 33–40, 2009.

[50] Li Xiong and Ling Liu. A reputation-based trust model for peer-to-peer e-commerce communities. In *IEEE International Conference on E-Commerce, (CEC'03)*, pages 275–284, 2003.

[51] Li Xiong and Ling Liu. Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):843–857, 2004.

[52] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. In *ACM SIGSOFT Software Engineering Notes*, volume 26, page 313, 2001.

[53] JBoss AOP. `http://www.jboss.org/jbossaop/`. Accessed March 2014.

[54] Kevin W Hamlen, Micah M Jones, and Meera Sridhar. Aspect-oriented runtime monitor certification. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 126–140. Springer, 2012.

[55] Kevin W Hamlen, Micah M Jones, and Meera Sridhar. Chekov: Aspect-oriented runtime monitor certification via model-checking (extended version). Technical report, Technical report, Department of Computer Science, University of Texas at Dallas (May 2011), 2011.

[56] Khaled Mahbub and George Spanoudakis. A framework for requirents monitoring of service based systems. In *Proceedings of the 2nd International Conference on Service-Oriented Computing*, pages 84–93. ACM, 2004.

[57] Apache BCEL: The Byte Code Engineering Library. `http://commons.apache.org/proper/commons-bcel/`. Accessed March 2014.

[58] Javasnoop. `https://code.google.com/p/javasnoop/`. Accessed March 2014.

[59] Aspect oriented programming implementations. `http://en.wikipedia.org/wiki/Aspect-oriented_programming`. Accessed August 2014.

[60] Azzam Mourad, Dima Alhadidi, and Mourad Debbabi. Cross-language weaving approach targeting software security hardening. In *6th Annual Conference on Privacy, Security and Trust, (PST'08)*, pages 87–98, 2008.

[61] The Java class file disassembler. `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html`. Accessed July 2014.

[62] Apache benchmark tool. `http://httpd.apache.org/docs/2.2/programs/ab.html`. Accessed July 2014.

[63] Java SE HotSpot. `http://www.oracle.com/technetwork/articles/javase/index-jsp-136373.html`. Accessed November 2015.

[64] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-Oriented Programming*. Springer, 1997.

[65] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of AspectJ. In *ECOOP Object-Oriented Programming*, pages 327–354. Springer, 2001.

[66] Bart De Win, Bart Vanhaute, and Bart De Decker. Security through aspect-oriented programming. In *Advances in Network and Distributed Systems Security*, pages 125–138. Springer, 2002.

[67] Viren Shah and Frank Hill. An aspect-oriented security framework. In *Proceedings of DARPA Information Survivability Conference and Exposition*, volume 2, pages 143–145, 2003.

[68] Zheng Li, Yan Jin, and Jun Han. A runtime monitoring and validation framework for web service interactions. In *Australian Software Engineering Conference*, 2006.

[69] Jocelyn Simmonds, Yuan Gan, Marsha Chechik, Shiva Nejati, Bill O'Farrell, Elena Litani, and Julie Waterhouse. Runtime monitoring of web service conversations. *IEEE Transactions on Services Computing*, 2(3):223–244, 2009.

[70] Luciano Baresi, Sam Guinea, Olivier Nano, and George Spanoudakis. Comprehensive monitoring of BPEL processes. *IEEE Internet Computing*, 14(3):50–57, 2010.

[71] Guoquan Wu, Jun Wei, and Tao Huang. Flexible pattern monitoring for WS-BPEL through stateful aspect extension. In *IEEE International Conference on Web Services, (ICWS'08)*, pages 577–584, 2008.

[72] Guoquan Wu, Jun Wei, Chunyang Ye, Hua Zhong, and Tao Huang. Detecting data inconsistency failure of composite web services through parametric stateful aspect. In *IEEE International Conference on Web Services, (ICWS'10)*, pages 68–75, 2010.

[73] Anis Charfi and Mira Mezini. AO4BPEL: An aspect-oriented extension to BPEL. *Journal of World Wide Web*, 10(3):309–344, 2007.

[74] Anis Charfi, Tom Dinkelaker, and Mira Mezini. A plug-in architecture for self-adaptive web service compositions. In *IEEE International Conference on Web Services, (ICWS'09)*, pages 35–42, 2009.

[75] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall, 2005.

[76] Luciano Baresi, Sam Guinea, and Liliana Pasquale. Self-healing BPEL processes with dynamo and the JBoss rule engine. In *International Workshop on Engineering of Software Services for Pervasive Environments: In Conjunction with the 6th ESEC/FSE Joint Meeting*, pages 11–20, 2007.

[77] Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, et al. The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 267–282. Springer, 2012.

[78] Kevin W. Hamlen and Micah Jones. Aspect-oriented in-lined reference monitors. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 11–20, 2008.

[79] Daniel J Dougherty, Claude Kirchner, Hélène Kirchner, and Anderson Santana De Oliveira. Modular access control via strategic rewriting. In *Computer Security (ESORICS'07)*, pages 578–593. Springer, 2007.

[80] Gabriela Gheorghe, Stephan Neuhaus, and Bruno Crispo. xESB: An enterprise service bus for access and usage control policy enforcement. In *Trust Management IV*, pages 63–78. Springer, 2010.

[81] Alexander Pretschner, Fabio Massacci, and Manuel Hilty. Usage control in service-oriented architectures. In *Trust, Privacy and Security in Digital Business*, pages 83–93. Springer, 2007.

[82] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[83] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[84] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. *Internet Society*, 2005.

[85] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[86] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[87] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[88] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazieres. Securing distributed systems with information flow control. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*, volume 8, pages 293–308, 2008.

[89] Srijith Nair, Patrick Simpson, Bruno Crispo, and Andrew Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16, 2008.

[90] Wei She, I-Ling Yen, and Bhavani Thuraisingham. Enhancing security modeling for web services using delegation and pass-on. In *IEEE International Conference on Web Services, (ICWS'08)*, pages 545–552, 2008.

[91] Wei She, I-Ling Yen, Farokh Bastani, Bao Tran, and Bhavani Thuraisingham. Role-based integrated access control and data provenance for SOA based net-centric systems. In *IEEE 6th International Symposium on Service Oriented System Engineering (SOSE'11)*, pages 225–234, 2011.

[92] Wei She, I-Ling Yen, Bhavani Thuraisingham, and San-Yih Huang. Rule-based run-time information flow control in service cloud. In *IEEE International Conference on Web Services, (ICWS'11)*, pages 524–531, 2011.

[93] Wei She, I-Ling Yen, Bhavani Thuraisingham, and Elisa Bertino. Security-aware service composition with fine-grained information flow control. *IEEE Transactions on Services Computing*, 6(3):330–343, 2013.

[94] Wei She, I-Ling Yen, Bhavani Thuraisingham, and Elisa Bertino. The SCIFC model for information flow control in web service composition. In *IEEE International Conference on Web Services, (ICWS'09)*, pages 1–8, 2009.

[95] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278. USENIX Association, 2006.

[96] Andrew C Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[97] Silvano Martello and Paolo Toth. *Knapsack Problems.* Wiley New York, 1990.

[98] List of knapsack problems. `http://en.wikipedia.org/wiki/List_of_knapsack_problems`. Accessed March 2014.

[99] Mhand Hifi, Mustapha Michrafy, and Abdelkader Sbihi. Heuristic algorithms for the multiple-choice multidimensional knapsack problem. *Journal of the Operational Research Society*, 55(12):1323–1332, 2004.

[100] List of knapsack problems. http://en.wikipedia.org/wiki/list_of_knapsack_problems. Accessed March 2014.

[101] Krzysztof Dudziński and Stanisław Walukiewicz. Exact methods for the knapsack problem and its generalizations. *European Journal of Operational Research*, 28(1):3–21, 1987.

[102] R Garey Michael and David S Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. WH Freeman & Co., San Francisco, 1979.

[103] Edward YH Lin and MC Chen. A dynamic programming approach to the multiple-choice multi-period knapsack problem and the recursive APL2 code. *Journal of Information and Optimization Sciences*, 31(2):289–303, 2010.

[104] Ch Ykman-Couvreur, Vincent Nollet, Francky Catthoor, and Henk Corporaal. Fast multidimension multichoice knapsack heuristic for mp-soc runtime management. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(3):35, 2011.

[105] Md Mostofa Akbar, Eric G Manning, Gholamali C Shoja, and Shahadat Khan. Heuristic solutions for the multiple-choice multi-dimension knapsack problem. In *Computational Science, (ICCS)*, pages 659–668. Springer, 2001.

[106] Rafael Parra-Hernandez and Nikitas J Dimopoulos. A new heuristic for solving the multichoice multidimensional knapsack problem. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 35(5):708–717, 2005.

[107] Shahadat Khan. Quality adaptation in a multi-session adaptive multimedia system: Model and architecture. *Ph.D. Thesis, Department of Electronical and Computer Engineering, University of Victoria, Canada*, 1998.

[108] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[109] Micrososft biztalk server. `http://www.microsoft.com/en-us/server-cloud/products/biztalk/`. Accessed August 2014.

[110] Tao Yu and Kwei-Jay Lin. Service selection algorithms for composing complex services with multiple QoS constraints. In *Service-Oriented Computing-ICSOC 2005*, pages 130–143. Springer, 2005.

[111] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for web services selection with end-to-end QoS constraints. *ACM Transactions on the Web (TWEB)*, 1(1):6, 2007.

[112] Chung-Wei Hang and Munindar P Singh. Trustworthy service selection and composition. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 6(1):5, 2011.

[113] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Francesco Lo Presti, and Raffaela Mirandola. Qos-driven runtime adaptation of service oriented architectures. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 131–140. ACM, 2009.

[114] Anja Strunk. Qos-aware service composition: A survey. In *IEEE 8th European Conference on Web Services (ECOWS'10)*, pages 67–74, 2010.

[115] Shi Yulu and Chen Xi. A survey on QoS-aware web service composition. In *3rd International Conference on Multimedia Information Networking and Security (MINES'11)*, pages 283–287, 2011.

[116] Md Mostofa Akbar, M Sohel Rahman, Mohammad Kaykobad, Eric G Manning, and Gholamali C Shoja. Solving the multidimensional multiple-choice knapsack problem by constructing convex hulls. *Computers & Operations Research*, 33(5):1259–1273, 2006.

[117] Chaitr S Hiremath and Raymond R Hill. New greedy heuristics for the multiple-choice multi-dimensional knapsack problem. *International Journal of Operational Research*, 2(4):495–512, 2007.

[118] Chung-Wei Hang, Anup K Kalia, and Munindar P Singh. Behind the curtain: service selection via trust in composite services. In *IEEE 19th International Conference on Web Services (ICWS)*, pages 9–16. IEEE, 2012.

[119] David Miller, Shon Harris, Allen Harper, Stephen VanDyke, and Chris Blask. *Security Information and Event Management (SIEM) Implementation.* McGraw Hill Professional, 2010.

[120] Ganglia monitoring system. `http://ganglia.sourceforge.net/`. Accessed October 2014.

[121] Netflix Chaos Monkey. `http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html`. Accessed March 2014.

[122] Sara Bouchenak, Gregory Chockler, Hana Chockler, Gabriela Gheorghe, Nuno Santos, and Alexander Shraer. Verifying cloud services: Present and future. *ACM SIGOPS Operating Systems Review*, 47(2):6–19, 2013.

VITA

VITA

Mehdi Azarmi obtained a B.Sc degree in 2005 and a M.Sc degree in 2006 in computer engineering at Amirkabir University of Technology in Tehran, Iran. He also obtained his Ph.D. degree from the Department of Computer Science at Purdue University under the direction of Professor Bharat Bhargava. He is affiliated with CERIAS, the Center for Education and Research in Information Assurance and Security at Purdue University. His research interests are information security, cloud computing, and large-scale distributed systems. He spent the summer of 2012 at Verisign Lab as a research intern. He worked as a principal software engineer at the advanced software division of EMC in Seattle, Washington from 2014 to 2016. He is a senior software engineer at Portworx in the San Francisco Bay area.