2007

Bootstrapping trust in service oriented architecture

Mahantesh Hosamani
Iowa State University

Bootstrapping trust in service oriented architecture

by

Mahantesh Hosamani

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Iowa State University

Ames, Iowa

2007

DEDICATION

To my teacher, who made me realize the real purpose of education.

TABLE OF CONTENTS

LIST OF TABLES v

LIST OF FIGURES vi

ACKNOWLEDGEMENTS vii

CHAPTER 1. Introduction 1

1.1 Thesis Outline 3

CHAPTER 2. Background 5

2.1 Service Oriented Architecture 5

2.1.1 Clark-Wilson Integrity Model 6

2.2 Trusted Platform Modules 7

2.2.1 Cryptographic Coprocessor 7

2.2.2 Cryptographic Keys 8

2.2.3 TPM Registers 9

2.3 TPM Usage Models 9

CHAPTER 3. Related Work 11

3.1 Secure o/s and cryptography techniques 11

3.2 Monitoring virtual machines and information flows 12

3.3 TCG based integrity measurement 13

3.4 Distributed Attestation Models 14

3.5 Modeling Non-functional Aspects of SOA 16

CHAPTER 4. Problem Definition 17

4.1 Detailed Problem Statement 18

4.1.1 Goal 18
4.1.2 Description 18
4.2 Hypothesis 19
CHAPTER 5. Approach 20
5.1 Implementation 22
5.1.1 Client 23
5.1.2 Service Provider 24
5.1.3 Trusted Third Party 25
CHAPTER 6. Evaluation 27
6.1 Experimental Setup 27
6.2 Violations 29
6.3 Overhead of Monitoring 30
CHAPTER 7. Case Study 31
7.1 Example Web Service - Shopping Cart 31
7.2 Experiment 31
7.2.1 Monitor detects trust violation 33
7.2.2 Detecting a compromised monitor 34
CHAPTER 8. Conclusions 36
8.1 Discussion 37
8.2 Future Work 37

LIST OF TABLES

6.1 Subjects for our Case Study 28

6.2 TPM Measurements for a Genuine and a Compromised Requirements Monitor 29

6.3 Overhead of Monitoring 30

7.1 Case Study: Comparison of TPM Measurements between a genuine and a compromised Monitor 34

LIST OF FIGURES

2.1 SOA Paradigm 6
2.2 TPM Component Architecture 8

5.1 Proposed Architecture 20
5.2 Interaction Diagram 23
5.3 Implementation of the Proposed Architecture 24

7.1 Example of Trust Violation 32
7.2 Output trace for the original service 32
7.3 Trace for compromised web service 33

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Hridesh Rajan for his guidance, patience and support throughout this research and the writing of this thesis. His creative insights and words of encouragement have often encouraged me and renewed my interest in research. His weekly research paper reading sessions presented a lot of opportunities for exploring newer domains in research and hitting upon synergistic ideas. I also thank him for sponsoring my travel to conferences which helped me broaden my perspectives and present my work to the research community.

I would like to thank my committee members Dr. Gurpur Prabhu and Dr. Samik Basu for their efforts and contributions to this work. I am very thankful to Harish Narayanappa for lending his technical skills during the implementation of this research project. I would like to thank Robert Dyer for proof-reading and for asking excellent questions. I want to acknowledge Rakesh Setty for offering his critical suggestions and inputs during our discussions. Additionally, I would like to extend my thanks to all other members of Laboratory of Software Design for offering constructive criticism and timely suggestions during research.

I am very grateful to Dr. Kasthurirangan Gopalakrishnan , Siddhartha Khaitan and Ankit Agrawal for their unwavering support and encouragement which enabled me to complete my research and thesis ahead of time. I am very grateful to my parents Mr. Linganagouda C. Hosamani and Mrs. Laxmi Hosamani, for extending their moral support and encouragement throughout the duration of my studies.

CHAPTER 1. Introduction

Service oriented architecture (SOA) is an architectural paradigm to promote reuse of systems and integration of subsystems to build new applications. The subsystems which are exposed as services can be composed or orchestrated to create new systems.

Though SOA promotes abstraction, loose coupling and interoperability, there are certain pitfalls with regards to security and trust. Web based services have the property of location transparency, i.e. client does not care about where the services are located. This gives rise to the *need to verify the trustworthiness of a remotely hosted computation.* It is important to see to it that the terms and conditions agreed to before a transaction carried out by the remotely hosted computation are not violated during or after the transaction. Among other things, this kind of verification is essential for regulatory compliance in the payment card industry; where sensitive customer data is increasingly at risk of being stolen or being used for adversarial purposes and there is a pressing need to protect the privacy and integrity of such data.

Specifying and verifying functional and non-functional requirements for service-oriented architectures had not received much attention until recently. Kuo *et al.* have discussed an approach for expressing and reasoning about functional requirements for service-oriented computing [25]. The focus of their approach is on facilitating a more concise representation of the message exchange protocols as a Boolean formula associated with each exchanged message, which in turn helps to verify whether a given message exchange is legal. On the other end of the spectrum are approaches to dynamically monitor web service compositions such as by Baresi *et al.* [8], Barbon *et al.* [7] and Mahbub and Spanoudakis [27]. Such approaches verify the service compositions for their syntactic and semantic accuracy.

The advantage of these approaches is that they are independent of the web service implementation. A significant disadvantage and the problem that this thesis addresses is that they do not allow verification of non-functional requirements such as integrity of the web services, information flow properties, etc. To verify such properties, a trustworthy verifier need to be deployed on the service provider's environment. I argue in this thesis that it is difficult to realize such a verifier using a *software-only* approach[1].

To overcome the above shortcomings, the key insight behind my thesis is to employ a hardware-based trust mechanism called Trusted Platform Module to bootstrap trust in a service-oriented architecture. Trusted Platform Module is based on the specifications provided by the Trusted Computing Group's initiative [2]. A related idea proposed by Sailer et al. also uses similar insights to measure the integrity of remote platforms [39]. This thesis advances upon the idea by proposing an architecture that serves to monitor remotely hosted web services and give an assurance about their integrity.

Implementing an architecture to conduct attestation of remotely deployed web services involved achieving secure interaction between three entities - the service oriented architecture, the TPM hardware and the requirements monitor. Since the root of trust in this architecture is derived from the TPM hardware, every proof of verification of the web service execution had to include the integrity measurement of the environment as reported by the TPM in the service provider's environment. Such a trustworthy measurement serves as the basis for giving an assurance to the client regarding the integrity of the web service.

We validated the implementation of the architecture by tracking the use of sensitive data in a web service execution by using the *requirements monitor*. Whenever there was a violation, it was detected by the system. If there was no violation, the system is capable of giving an assurance about the transaction to the client.

This work demonstrates the feasibility of implementing an architecture that establishes trust in service oriented architecture. The implementation has been successfully tested on standard web services available with the Apache Axis distribution. Using this architecture, it is possible to give an assurance or detect a requirement-violation for every execution of a

[1]These ideas were first proposed and discussed in our publications [19, 20]. This thesis is a consolidation of these previous works.

remotely deployed web service. The results show that it is feasible to effectively monitor web services in a service oriented architecture for non-functional requirements such as privacy and integrity with a short time-overhead.

1.1 Thesis Outline

The rest of this thesis is organized as follows:

Chapter 1: The primary driver for Service Oriented Architecture (SOA) implementation is *change*. SOA essentially consists of services which are units of individual functionalities designed according to certain functional specifications or requirements. These services are deployed on remote servers and it is a challenge to monitor such services for non-functional requirements such as security, integrity and data confidentiality. This chapter presents a brief overview of the research carried out in the direction of monitoring such non-functional requirements.

Chapter 2: An introduction to Service Oriented Architecture and its applications are presented. This chapter also covers the component architecture of Trusted Platform Module and its usage models.

Chapter 3: Existing architectures and security paradigms, along with the comparison to the proposed architecture are discussed in this chapter. The chronology of development of these architectures, models and paradigms is traced. The ideas are classified into various categories based on similarities of concepts and platform-architectures.

Chapter 4: The motivation for monitoring non-functional requirements in web services is discussed. This chapter also elaborates the problem we address and the goal of this research. This is followed by the hypothesis of building a layer of trust into Service Oriented Architecture based on the Clark-Wilson model.

Chapter 5: Our approach towards guaranteeing trust in SOA involves the use of Trusted Platform Modules (TPMs) to accurately monitor and report the status of the Requirement Monitoring software, which in turn oversees the secure execution of the web service. The proposed SOA architecture includes an additional interface called the *trust negotiation and* *verification interface* which establishes trust between the communicating parties in a SOA. Trust negotiation happens once and the reposed trust can be verified by the client for every transaction through the trusted third party. This chapter describes the proposed system architecture, components within the proposed architecture and its implementation in detail.

Chapter 6: This chapter demonstrates the feasibility of implementing the proposed architecture in the context of a Service Oriented Architecture using a Trusted Platform Module. The proposed architecture is used to evaluate web services available with the standard Apache Axis distribution. The results demonstrate the effectiveness of the proposed architecture to verify the integrity of web services in spite of incurring a short time-overhead.

Chapter 7: In this chapter, a representative web service - Shopping Cart is selected as a candidate for explaining the detailed evaluation procedure that was adopted to evaluate every web service of the Apache Axis distribution. A comparison of the execution traces of the genuine and compromised web service is also shown.

Chapter 8: The conclusion of the thesis is presented here. Some of the security issues with Service Oriented Architecture (SOA) and how the proposed architecture can be used to address important problems like that of monitoring of non-functional requirements are discussed. A discussion on the future directions of research involving the use of program analysis techniques and virtual-machine based monitoring to improve the efficiency of the proposed architecture is also included.

CHAPTER 2. Background

A breif discussion on Service Oriented Architecture (SOA) and Trusted Platform Modules (TPM) is presented in this chapter.

2.1 Service Oriented Architecture

Service Oriented Architecture consists of services. A service is a well-defined and a self-contained function that does not depend on the context or state of other services. Every service has a published interface, often in the form of a WSDL (Web Services Definition Language) file [12]. A service oriented architecture is a collection of such services. A service oriented architecture essentially defines three type of entities: service implementation (or providers), service mediation (or brokers), and service consumption (or clients) [15, 30, 31].

Figure 2.1 illustrates the *find-bind-execute* paradigm used in SOA. In this paradigm, service providers register their services in a public registry called a *service broker*. The clients use this broker to find services that match their criteria. The broker provides a contract between the client and the service provider. The services may communicate with each other either by exchanging data or they may coordinate with each other to perform some complex operation. In a SOA, the resources in a networked environment are all exposed as services that can be made use of by a client on the network through their published interfaces without the knowledge of their implementation details. The aim of this work is to guarantee the integrity of such implementations of web services.

Service oriented architectures (or service oriented computing paradigm) promote abstraction, loose coupling and interoperability of clients and services [31]. The key idea is to introduce a published interface (often in the form of a description written in web services definition lan-

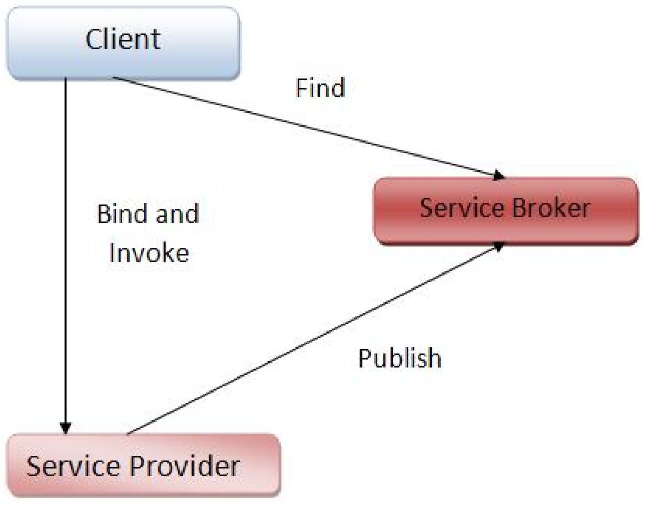

Figure 2.1 SOA Paradigm

guage, WSDL [12]), which acts as a basis for communication between three types of entities: service implementation (or providers), service mediation (or brokers), and service consumption (or clients), that often follows the sequence publish-find-bind-execute to discover and use services [31]. The published interface describes the functional requirements for co-ordination between service implementations and clients. Every service implementation must satisfy its functional requirements. For example, a published interface for a location-based hotel finding service may expect clients to provide GPS co-ordinates in a specified format and expect the service implementation to produce the address of the nearest hotel as a string to those GPS co-ordinates. The specification of input (GPS co-ordinates) and output (string containing the hotel's address) describe the functional requirements for this service.

2.1.1 Clark-Wilson Integrity Model

To guarantee integrity of web service implementations, we use Clark-Wilson Model [13] in SOA. Clark Wilson integrity model is a foundation for specifying and analyzing integrity of any computing system. Clark-Wilson integrity model formalizes the concept of information integrity. Clark-Wilson model particularly emphasizes that the implementer of the transaction and the certifier of the transaction are essentially different entities. According to the model, any well-formed transaction should transition a system from one consistent state to another consistent state. To monitor this, there has to be a mechanism that transparently reports the state of the service provider's system from time to time. Such a mechanism should not be vulnerable to any kind of tampering.

2.2 Trusted Platform Modules

Over the past few years, the computer industry has come up with many initiatives to guarantee security, integrity and confidentiality of data through innovative hardware-based architectures. A consortium of key industry players, Trusted Computing Group (TCG) [42], came up with the specifications for a TPM with such a goal. The TCG vision was that this rudimentary TPM supported trust can be bootstrapped into a higher level trust through some software trust architecture or design principle. Another popular initiative is the Next-Generation Secure Trusted Computing Base (NGSCB) [29]. Hardware vendors are moving towards installing TPM on every computer that ships.

A Trusted Platform Module (TPM) is a trusted agent co-processor within a remote computing platform which derives its root of trust from its manufacturer or a delegated trusted third party [42]. A TPM can be trusted to perform certain actions truthfully despite being an integral part of a potentially malicious or compromised system. In other words, it is our trusted ambassador in a friendly or hostile foreign territory. The TPM hardware, firmware and the software provides a platform root of trust. A TPM can extend its trust to higher layers of the system by building a chain of trust starting from the hardware and subsequently linking one layer to the next.

Figure 2.2 lists the components that constitute a TPM [2]. The I/O component manages the information flow over the communication bus.

2.2.1 Cryptographic Coprocessor

The cryptographic coprocessor of TPM implements cryptographic functions which are executed within the TPM hardware. Hardware or software entities outside the TPM have no

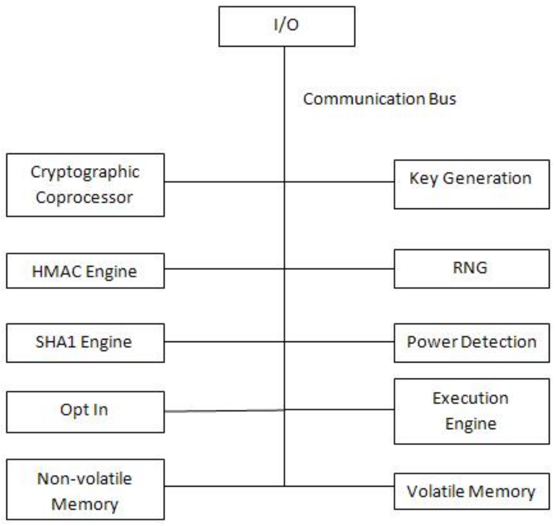

Figure 2.2 TPM Component Architecture

access to the execution of these functions. A TPM also contains a RSA accelerator to perform 2048 bit RSA encryption and decryption. The TPM uses RSA algorithm for signature operations on internal and external items. There is also an engine for computing SHA1 hash for small pieces of data within the TPM. This SHA1 interface is exposed to the software entities outside the TPM to support taking of measurements during the platform boot phases.

A Random Number Generator (RNG) is the source of randomness in TPM. It provided for generating keys for key generation, generation of nonces and for randomness in signatures. This randomness capability is protected from external access. A TPM manages its power state as well as the power state of the platform through the Power Detection component. The Opt-in component allows the TPM to be turned on/of or activated/deactivated. The Execution Engine executes the commands received from the I/O component of the TPM.

2.2.2 Cryptographic Keys

Every TPM is identified by a built-in key called the *Endorsement Key*, which is included in it by the manufacturer . The key size is 2048 bits. The trust that one reposes in a TPM comes from the fact that this key is unique and is protected at all times in the TPM. An *Endorsement Certificate*, which contains the public key of the Endorsement Key, certifies this property. This key can be used by the owner to anonymously confirm that the *identity keys* were generated by the TPM in his system. In essence, every computer has a unique identity which cannot be repudiated. This can serve to be a fool-proof identity for every user. The TPM manufacturer provides a certificate for the Endorsement Key.

There is another kind of keys related to Endorsement Key called the *Attestation Identity Keys* (AIK). These keys are used by a *privacy CA* to present different AIKs to different remote parties to enable the system to hide its platform identity from other systems.

The TPM can be used to store three kinds of certificates [6].

- **Endorsement Certificate:** Attests that a particular platform configuration is genuine. This contains the public part of Endorsement Key.
- **Platform Certificate:** Attests that the security components of the platform are genuine. This is provided by the platform vendor.
- **Conformance Certificate:** This can be provided by a third party to certify the security properties of the platform.

2.2.3 TPM Registers

The TPM also has a set of registers called *Platform Configuration Registers*(PCR) which can be used to store the 160-bit hash values obtained using the SHA1 hashing algorithm of the TPM. The hardware ensures that the hash value of any PCR can be changed only by encrypting the new data over the previous hash value of the PCR. In this way, PCRs can be used to indelibly record the history of the machine since the last reboot. The PCRs are cleared off at the time of system reboot. PCR values are used to decide whether a system can be trusted.

2.3 TPM Usage Models

Bajikar [6] describes the application of TPM in protecting confidential data and certifying credentials of the platform. Accordingly, three important usage models of the TPM have been described.

- **Hardware Protected Storage:** The TPM can be employed to protect sensitive data of the user by encrypting the secret data in such a way that it can only be decoded on a specific hardware that contains the necessary private key.
- **Information Binding:** Certain critical data can be bound to a particular platform in such a way that it is accessible only if the conditions specified during the binding are met. Data will be rendered inaccessible if it is migrated to a different platform.
- **Platform Authentication:** Attestation Identity Keys are always bound to the platform. These can be used to authenticate the user and the platform.

Critics of TPM claim that TPMs will have a huge impact on user privacy. Service providers with commercial interest will try to misuse the power of TPM by introducing stricter controls and by eliminating user-anonymity. Anderson et al. present a detailed discussion on the recent controversies and the benefits of trusted computing in [3]. Sadeghi et. al. have proposed a hybrid kernel architecture called PERSEUS [33], to reduce the scope of the strict controls of the TPM and to restrict the controls to wherever its necessary and applicable. In this architecture, the client operating system provides the user with a common user interface which is backward compatible, to reuse all uncritical standard applications and services such as file system and network operations. In [37] they extend the PERSEUS kernel architecture to support TCPA (Trusted Computing Platform Alliance) and Microsoft Palladium.

CHAPTER 3. Related Work

In this section we trace the chronology of a variety of techniques and paradigms that were developed to protect sensitive data and, to secure virtual machines and operating systems.

3.1 Secure o/s and cryptography techniques

Ever since the 1970s, efforts have been made to produce secure operating systems [40] as a basis for secure computing. Any system can be thought of as consisting of many layers of abstractions. The integrity of a system is built recursively through a chain of integrity checks starting from the lowermost level of abstraction. Each level is checked for integrity before passing the control to the next higher level. In 1997, Arabaugh et al. proposed an architecture for secure and reliable bootstrapping called AEGIS [5]. In AEGIS, the integrity checks begin from the power-on and continue till the control is handed over to the operating system. AEGIS modifies the boot process so that all executable code is verified using digital signatures prior to its execution. Here, the chain of trust begins from the software loaded in BIOS and PROM boards. AEGIS guarantee that the system initializes to a secure state.

Another aspect related to security of the sensitive data is verifying untrustworthy applications before executing them locally. This involves establishing trust between the code producer and the host. Cryptography [9] can be used to ensure that the code was produced by a trusted source. Occasionally, even trusted sources may act maliciously because of human errors. Since this system relies on a personal authority to be impeccable, it is not very strong. In 1997, Necula [28] introduced the concept of proof-carrying code. This is a mechanism by which the host system can determine if an untrustworthy application is safe to be executed within its environment. This process consists of a *security policy* which is defined by the code consumer.

The safety policy has a set of safety rules and the interface. The safety rules record a list of authorized operations and their safety preconditions. The interface describes the invariants that hold during and after invoking functions provided by the untrustworthy code. The code is certified by the producer and is validated by the code consumer. The code validation is done only once irrespective of the number of executions, using an algorithm which the host trusts. The major advantages of this approach are that it requires lesser time and little user interaction.

Microsoft has incorporated a feature called Secure Startup starting with Windows Vista [29]. Secure Startup has the capability to ensure that the PC running Vista starts in a known good-state. AEGIS cannot distinguish fake hardware from the genuine one. If the booting process is not sequential, certain non-trivial changes have to be made to the architecture.

3.2 Monitoring virtual machines and information flows

In 2003, Grafinkel et al. proposed Terra, a virtual-machine based platform for trusted computing. In Terra, a virtual machine monitor was used to simultaneously partition the hardware into independent, isolated virtual machines. The software stack of each virtual machine could be tailored to meet the security requirements of the software running on that virtual machine. However, it is not possible to selectively measure individual software. The ever increasing number of device drivers pose a formidable challenge to implement the virtual machine monitor. Terra does not address the issue of loading untrustworthy drivers. Unlike AEGIS, Terra does not start from a secure boot process.

There are many ways and means to enforce policies such as confidentiality and security on the end-to-end behavior of a computing system. Such methods are broadly classified as *Information Flow Mechanisms.* Other than carrying out a rigorous analysis on the system as a whole to prove that it enforces the specified security policies, Information Flow Mechanisms also take into consideration the possibility of supplying malicious inputs to the program so that it terminates abnormally. It is then verified if confidential information can be extracted from the exception trace.

Sablefeld et al. address such issues through language-based techniques for specification and enforcement of security policies in [36]. The limitation of this approach is that the security policies can only be specified by the programmer. The user of the software has no say in it. Identifying such short-comings, Vachharajani et al. proposed RIFLE [44], a user-centric run-time information flow architecture. Information flow systems such as this allow untrustworthy applications to access confidential data but prevents the data from getting leaked to other programs or covert channels. The authors claim that RIFLE can be used to enforce user-defined security policies on any program through a security-enhanced operating system. The program binary is translated from the conventional *Instruction-Set Architecture* (ISA) to an *Information Flow Secure* (IFS) architecture. This translated program is executed on a hardware designed for information-flow tracking. The goal is to verify if the program contains only legal flows, which is defined by the user in the security policy. Such an architecture is very useful if the user wants to be certain that the program running on his/her machine is not propagating any confidential local data, that the user is unaware of. It is difficult to apply this technique without major changes in the context of a service oriented architecture because the web service implementation program runs elsewhere rather than locally.

3.3 TCG based integrity measurement

In 2004, Sailer et al. proposed a TCG based Integrity Measurement Architecture for Linux [39]. This architecture made use of a Trusted Platform Module (TPM) hardware to store the integrity measurements of the system using the SHA1 hash function module of the TPM hardware. Unlike AEGIS, this system only takes measurements and does not have a recovery process. Also, this system can take selective measurements of the software to create a representative evidence that can be interpreted by the remote party.

The purpose of this architecture is to present an ordered list of measurements to a remote party. The remote system determines the integrity of the attested system by reconstructing the image of the attested system's software stack on the local system using these measurements and then by applying the security policy on the local software stack. To establish mutual trust, this process has to be carried out on both sides involved in the transaction [38]. This was implemented by instrumenting the Linux kernel to create measurements and to store them in the TPM hardware to protect against compromised systems. This architecture takes measurements of the kernel modules, executables and shared libraries, configuration files and other important files before they are loaded on the system. The advantage of this architecture is that it could verify integrity of a system up to it's application layer (web server).

However, the process of mutual attestation is quite complex involving recreating the image of the other party on the local system based on the measurements obtained and then applying a security policy to it. The task of taking measurements is implemented by making modifications to the Linux kernel code. In case of online transactions, common users may not have the Linux operating system. In a majority of the cases, the two communicating parties may not have the same operating system in their environments. This makes it difficult to recreate the image locally based on the measurements sent out by the other party. Our architecture is designed to address these issues.

Haldar et al. discuss about the broad problems with remote attestation in [18]. According to them, the most critical shortcoming regarding remote attestation is that it is not based on program behavior. In our architecture, this problem is solved by having the requirements monitor report the program behavior. Another problem they point out is that remote attestation is static and inflexible. Though this is true, it does not affect the viability of our architecture because we are not directly measuring the applications which may change frequently, but all that we monitor is the specific requirements of the applications that are not supposed to change.

3.4 Distributed Attestation Models

Yoshihama et al. proposed WS-Attestation [47], an attestation architecture on web services framework. This architecture leverages TCG technologies and allows establishing trust among distributed parties. WS-Attestation is built on top of existing web services standards. Four kinds of attestations are proposed - direct attestation, pulled validation, pushed validation and delegated attestation. This model is similar to our architecture, in that, a third party validates or performs attestation on behalf of the requester. They use an *integrity database* as a infrastructure for supporting attestation. This database stores the hash of the packages installed in the operating system. Further, the authors have tried to map this model to WS-Trust [4] by implementing the challenge-response protocol through message exchanges. The goal of this research is to validate the platform on which the web services are running. WS-Attestation can report errors if the remote platform is affected by viruses or other malware. The architecture proposed in this thesis work goes beyond validating the remote platforms. Using the proposed architecture, not only the platforms can be validated, but also the web services themselves can be monitored for integrity violations and compromises. Our work mainly concentrates on monitoring the key requirements of web services to detect compromises.

In November 2006, Katsuno et al. proposed a new model of a distributed coalition, called Trusted Virtual Domain(TVD) [24] for establishing trust among components in a heterogeneous distributed computing environment. TVD supports distributed mandatory access controls whose security policies can be different in each domain. A TVD can enforce the security policies on any components that wish to join that domain. They propose a layered negotiation approach for negotiating trust. This design makes use of Trusted Computing Base (hardware) as the lowest layer. Assurances of Trusted Components and TVD agent are achieved by chains of trust which derive the root of trust from the TCB. Attestation occurs in two stages - local and global. The global attestation verifys primitives generated by the TCB and the local attestation verifies component-specific parts depending on the usage-scenarios.

Park et al. present an attack resilient trust model to capture the trustworthiness of the web service in [32]. Unlike our approach, this approach depends on the cumulative measurements of trust through multiple requests and responses exchanged among the participants.

Another approach towards achieving trust is aglet [34]. An aglet is a java object with a code component and a data component. The key idea here is to use these mobile agents to preserve privacy. An aglet consists of two distinct parts: the aglet core and the aglet proxy. The aglet core contains all the internal variables and methods. It provides interfaces through which the environment can make use of the aglet or vice versa. The core is encapsulated with an aglet proxy which acts as a shield against any attempt to directly access the private variables and methods of the aglet. This aglet proxy can be programmed to enforce local privacy requirements on the site of the remote entity. Aglets are deployed into aglet servers, which enforces the requirement of the security model. A key problem with aglets is that the integrity of aglets depends on the integrity of aglet servers, which cannot be guaranteed in an untrustworthy environment. However, our architecture can be used to ensure the integrity of the aglet server, which would then provide a basis of integrity for aglets.

3.5 Modeling Non-functional Aspects of SOA

Wada et al. proposed a UML profile to graphically model non-functional aspects in SOA so that they are incorporated in the development phase [45]. This UML profile includes certain key model elements of service oriented architecture such as *service, message exchange, message, connector and filter*. This model driven development (MDD) paradigm for addressing non-functional concerns such as security and integrity in the service oriented architecture is an encouraging step for developing a secure service oriented architecture.

Canfora et al. have presented a detailed analysis of the fundamental issues and solutions related to various perspectives of testing a service-centric model in [10]. These perspectives are analyzed considering the needs of the service provider, the system integrator, the third party certifier and the user. The authors profess that making a service-centric system capable of self-testing helps overcome issues such as unpredictable response time and availability. This idea of self-testing was implemented by using the *trust analyzer* and by extending the concept to include trust as one of the issues in a service-centric system. The proposed architecture avoids wastage of resources because it does not force the service provider, the trusted third party and the user to make any radical changes in the existing architecture. The only initiative that has to be taken for guaranteeing trust is to use the TPM hardware which is already a part of the system.

CHAPTER 4. Problem Definition

In service-oriented architectures, services are often performed on machines that are not owned or operated by the clients. Composition of services may happen on an entirely different machine all together. In case of a location-based hotel finding service, the service may expect clients to provide GPS co-ordinates in a specified format and expect the service implementation to produce the address of the nearest hotel as a string to those GPS co-ordinates. The specification of input (GPS co-ordinates) and output (string containing the hotel's address) describe the functional requirements for this service. Suppose the service is to be monitored for the following requirements.

- R1: (Functional Requirement) The response given by the location-based hotel finding service shall be the closest hotel to the GPS co-ordinates specified by the client
- R2: (Non-functional Requirement)The service shall not persist the GPS co-ordinates supplied by the client

For R1, it is sufficient to observe or test the interface of the service. On the other hand, to validate requirements such as R2, it may not be sufficient to validate just the external interface.

The validation for most non-functional requirements may only come from a monitor that is executing in the same domain as the service implementation and that can validate — by observing the running service implementation — that the requirements such as R2 are indeed satisfied. The design and development of these monitors is a widely studied problem in requirements monitoring literature (e.g. see [16, 17, 26, 35]). Nevertheless, the key question remains; *in a (possibly) untrustworthy domain who guarantees the integrity of the monitor? In other words, who monitors the monitor?*

4.1 Detailed Problem Statement

4.1.1 Goal

The goal of our approach is as follows: given a set of service specification (S), a set of service implementation (I), a monitor that is capable of detecting deviations in the execution of the service implementation from its specification ($M : S \times I \rightarrow \{true, false\}$) running in a trusted environment, and a monitor that is similarly capable, but may be running in an untrustworthy environment ($M' : S \times I \rightarrow \{true, false\}$), how can we validate that $M \equiv M'$ is always true.

4.1.2 Description

To give the reader an idea of the problem with verifying a monitor in an untrustworthy environment without a root of trust, let us for a moment assume that a validation mechanism $V' : M \times M' \rightarrow \{true, false\}$ exists. Now in order to answer this validation question, there must be a part of V' running in the same untrustworthy environment that can observe M' to compare it with M. If not, V' will depend on the untrustworthy environment to observe M', which in turn may mask the true responses of M' with expected responses for M thereby invalidating the premise that V' exists. On the other hand, if some part of V', say $\delta V'$ is running in the same untrustworthy environment to observe M', we will need another monitor to verify that the integrity of $\delta V'$ is not compromised, which will need to be verified again, *ad infinitum*. In summary, V' may not exist.

Using standards such as WS-Security [23] and WS-Trust [4] or proposals such as that by Skogsurd et al. [41], we can address the issue of security-token interoperability and secure messaging within SOA. But, these standards are not sufficient to guarantee trust in an untrustworthy environment. Existing software solutions mainly concentrate on encrypting files, messages or other data. Such solutions have an inherent weakness - storage of passwords or keys. Secure algorithms with large keys are difficult to remember and are hence stored on the disk. Access to such disks may be password-protected. But, a sophisticated hacker may be able to crack such a security mechanism. [1]

We propose to use a hardware-based mechanism as a root of trust for such validation mechanisms. Along with a hardware-based mechanism, these security standards can be used to enhance security and guarantee trust in SOA. Let us consider the example described in the previous paragraph. In this example, if we could be sure that there exists a $\delta V'$ such that we do not need another monitor to verify its integrity, $\delta V'$ would make V' realizable. Fortunately, recent research results have shown that realization of such hardware-based root of trust is possible in the form of a *Trusted Platform Module* (TPM) [39, 38]. TPM is a co-processor that is now being shipped with every CPU from major processor vendors and is therefore available broadly. In this work, we describe an architecture based on TPM to validate the integrity of a runtime requirement monitor, which will in turn facilitate trusted services.

4.2 Hypothesis

The main goal of this research is to preserve trust among the entities in a service oriented architecture. The current specification for web services in a service oriented architecture gives a lot of flexibility and freedom to the service providers and does not prevent them from implementing the services as they like. Currently, there is no fool-proof mechanism to verify the service provider's claim. Therefore, an approach is needed to guarantee the integrity of the web service. We employ the Clark-Wilson model, which was described in Chapter 2, to monitor the integrity of the requirements monitor which in turn monitors the web service.

Following the Clark-Wilson model, we proposed certain key additions to the standard SOA in the form of a new interface called *trust negotiation and verification interface* [21]. The purpose of this interface is to provide an abstraction for the clients to negotiate desired integrity levels and for brokers to verify the conformance of the service implementation with the desired specification. The interface also allows broker to communicate with its trusted agent, the trusted platform module, and with service specific trust monitor in the service providers domain. The role of the trusted platform module is to periodically validate the integrity of the trust analyzer that in turn validates the conformance of the service implementation with the service specification.

CHAPTER 5. Approach

In the traditional service oriented architecture, there are three main entities: the service provider, the service broker and the client. The client contacts the service broker with a request and the broker finds a suitable provider to service the request and then directs the requester to that service provider. The client then binds to the service provider. Though this model offers a lot of flexibility for the entities involved, it lacks a layer or an interface that can inspire trust.

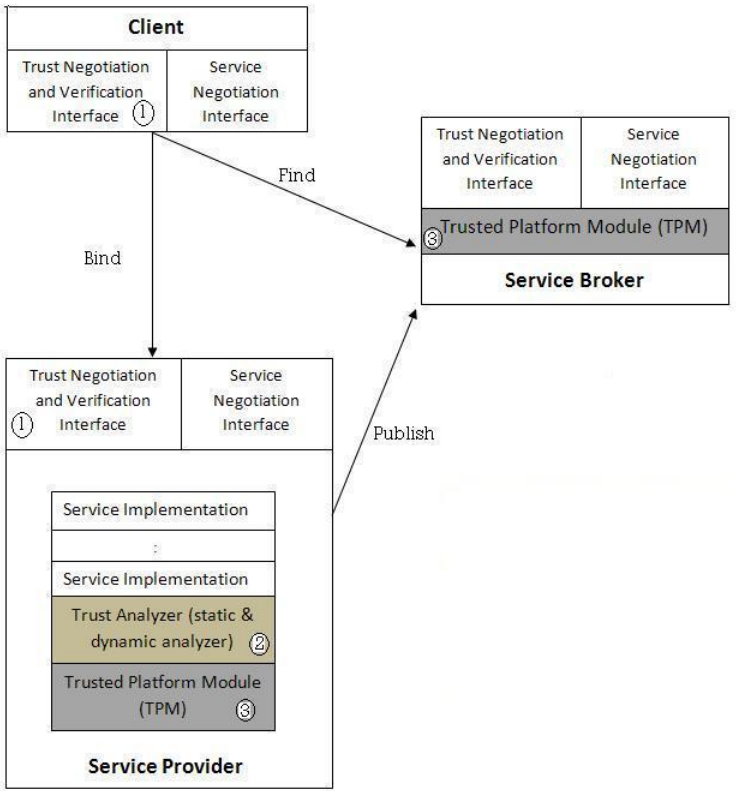
Client
Trust Negotiation and Verification Interface ①
Service Negotiation Interface
Find
Bind
Trust Negotiation and Verification Interface
Service Negotiation Interface
③ Trusted Platform Module (TPM)
Service Broker
Publish
① Trust Negotiation and Verification Interface
Service Negotiation Interface
Service Implementation
:
Service Implementation
Trust Analyzer (static & dynamic analyzer) ②
Trusted Platform Module (TPM) ③
Service Provider

Figure 5.1 Proposed Architecture

To provide for the missing trust, we propose certain key additions to the standard SOA in the form of a new interface called *trust negotiation and verification interface* as shown in

Figure 5.1. The purpose of *trust negotiation* interface is to facilitate negotiation of the desired level of trust between the trusted third party and the service provider. Negotiation culminates in a contract between the two parties. For every transaction, the trusted third party expects the service provider to provide a proof of execution of this contract. There are many ways of establishing such contracts. One of the ways is for both the parties to agree on monitoring certain variables carrying sensitive data. To fulfill such a contract, the service provider has to send the program traces containing the definition and use of such variables during the execution of the web service.

After the *trust negotiation* phase, the trusted third party receives the report from the service provider as per the contract. Now, the third party has to determine whether the report thus received is genuine or not. This is because the reports can be either faked or replayed for the purpose of covering the truth. This verification is carried out by the *trust verification* interface. To prevent deception, the third party has a trusted agent on the service provider's environment called the Trusted Platform Module (TPM). The *trust verification* interface communicates with the TPM on the service provider's environment and obtains a report about the integrity of its local environment from this TPM. Such a report can include an immutable integrity measurement of the local architecture along with a time stamp to indicate the freshness of the measurement.

The key insight that enables us to have a *trust verification* interface is the fact that the integrity measurement carried out by the TPM in the local environment of the service provider cannot be changed, even by the owner of the system. However, such a measurement can be read by anyone. This measurement can be read by the *trust verification* interface of the trusted third party to accurately determine the integrity of the service provider. The cryptographic capabilities of the TPMs can be leveraged to establish a secure communication channel between the service provider and the trusted third party. This channel can be used to convey the integrity measurements to the trusted third party.

The service broker is an ideal candidate for playing the role of the trusted third party. A *requirements monitor* that verifies whether the service implementation on the service provider's side conforms to its quality and integrity requirements can be used for servicing the contract which was established during the *trust negotiation* phase.

At the end of every transaction, based on the integrity measurements received, the trusted third party can either send an assurance or a notification of violation to the client. In this way, the client can trust the web service for its integrity.

Revisiting the hotel-finding service discussed in Chapter 4, suppose the client wants to know if the service fulfilled the non-functional requirement of not persisting the GPS coordinates supplied by the client. In this case, the trusted third party would have negotiated such a contract with the service provider. For this the contract to be enforced, the service provider has to have a program that monitors the operations carried out on the variables which handle the GPS coordinates. After every invocation of this web service, the trusted third party receives a report of integrity of such a program and the platform on which the program is running from the service provider's TPM. Based on this measurement, the trusted third party can either provide an assurance of genuineness to the client or notification of a violation to the client .

5.1 Implementation

The implementation consists of two client-server setups running parallely on the service provider and on the trusted third party. The process of monitoring is launched by the client who requests the trusted third party to monitor a web service before invoking it.

In this architecture, we assume that the operating system on the service provider's environment is secure, implying that the service provider will not be able to change to monitoring software without knowledge of the TPM in the system. For example, the approach proposed by Sailer et al. to secure the operating system kernel can be used [38]. Section 5.1.1 shows how the monitoring of a service is initiated by the client. Section 5.1.2 and section 5.1.3 list out the interactions between the clients and the servers on the service provider and on the trusted third party. The communication between the components of the proposed architecture is shown using the interaction diagram in Figure 5.2. The concrete implementation on the system is depicted in the system diagram in Figure 5.3

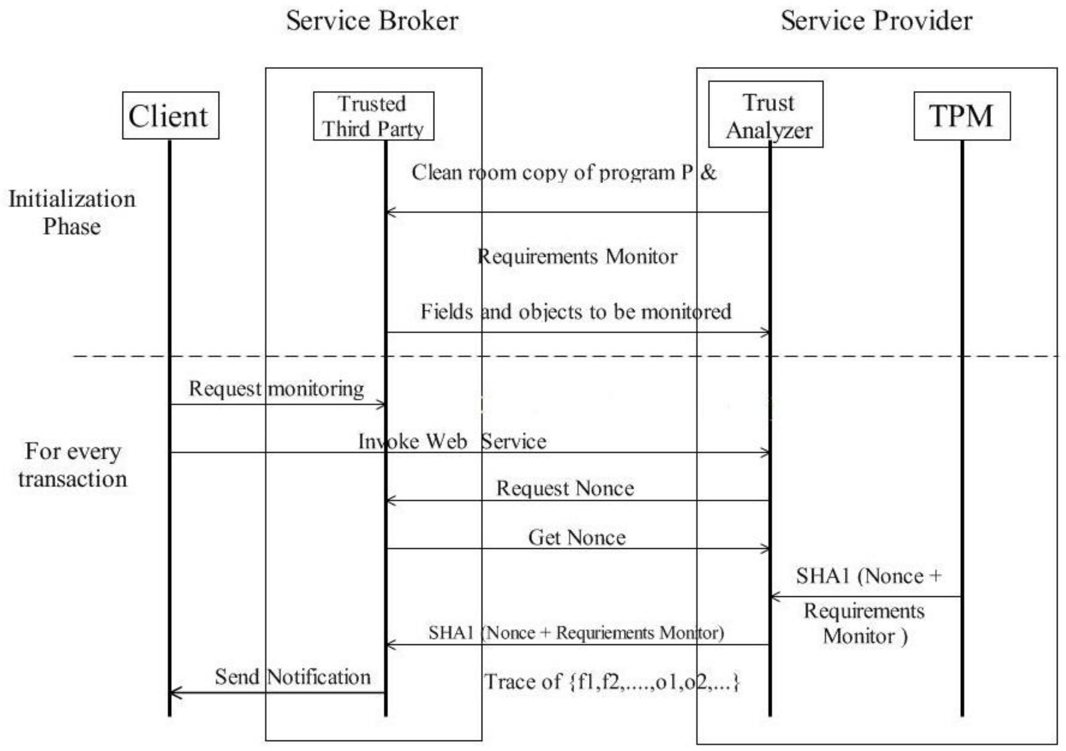

Figure 5.2 Interaction Diagram

The initialization phase is carried out under the supervision of a trusted authority. During this phase, the authentication server identifies the requirements to be monitored. We emulated the requirements identification process which consists of determining variables and methods dealing with data labeled as *sensitive*, by using Kaveri [22], a tool for program slicing. In future, we plan to have the requirement specifications as a part of the web service interface itself, thereby decoupling the process of requirements identification, which is currently tied to the implementation of the web service.

5.1.1 Client

On the client's side:

- Notify the trusted third party before the transaction.
- Invoke the web service.

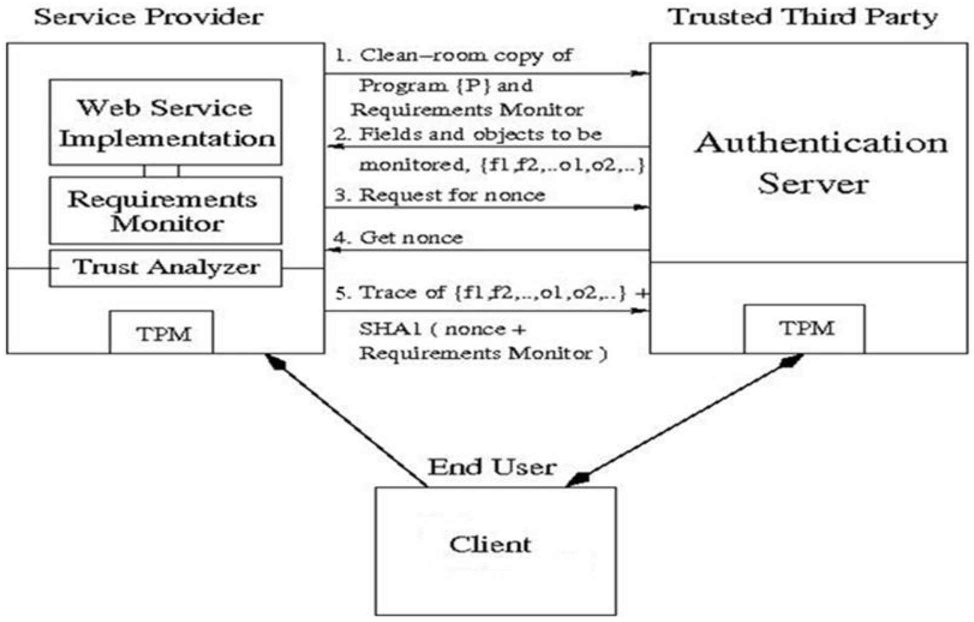
Service Provider
Trusted Third Party
Web Service Implementation
Requirements Monitor
Trust Analyzer
TPM
1. Clean–room copy of Program {P} and Requirements Monitor
2. Fields and objects to be monitored, {f1,f2,..o1,o2,..}
3. Request for nonce
4. Get nonce
5. Trace of {f1,f2,..,o1,o2,..} + SHA1 (nonce + Requirements Monitor)
Authentication Server
TPM
End User
Client

Figure 5.3 Implementation of the Proposed Architecture

- Wait for an assurance or notification from the trusted third party.

5.1.2 Service Provider

The Trust Analyzer lets the TPM monitor the *requirements monitor* which in turn monitors the execution of the web services at runtime. The TPM on the service provider's system uses AIK (Attestation Identity Key) to encrypt data in order to to make the interactions secure. AIK is a special purpose asymmetric signature key created by the TPM manufacturer, the private portion of which is non-migratable and protected by the TPM. Since the private part of the TPM's AIK cannot be retrieved by any user, the decryption of the data has to be done only on the trusted third party's machine using the private part of it's AIK.

Initialization phase on service provider's side:

- Accept the requirements to be monitored.

For every transaction, the following actions are carried out by the trust analyzer:

- Get a nonce from the authentication server of the trusted third party.
- Generate a trace for the specific web service implementation program using the *requirements monitor*.
- Using the local TPM, compute the SHA1 hash of the software stack up to the *requirements monitor* including the nonce.
- Encrypt the hash using the public part of AIK of the TPM of the trusted third party.
- Send the execution trace, nonce and the execution trace to the trusted third party.

5.1.3 Trusted Third Party

The trusted third party hosts an *authentication server*. For every transaction, the authentication server generates a unique nonce to guard against replay attacks. A nonce is a random number that is generated only once and is included in all interactions of a particular session to prove the freshness of data. It is to be noted that the trace is not being verified for an exact equivalence. This will lead to false positives because a program can be modified for purposes such as bug-fixing, without violating the specifications. Therefore, we programmatically check for violation of specific properties listed before. If any of the system, configuration or library files up to the *requirements monitor* are even slightly tampered, there will be significant variations in the final SHA1 hash value. It takes about 2^{69} units of time to find SHA1 collisions according [46], implying that collisions are very rare. Hence, these variations can be detected easily.

Initialization phase on the trusted third party:

- Identify the requirements to be monitored.
- Install the *requirements monitor* and on the program and generate a trace for the requirements determined above.

- Using TPM, measure the software stack up to the *requirements monitor* and store this measurement for future reference.

For every transaction, the following are done by the authentication server:

- On receiving a request from the client for monitoring, send a nonce to the service provider.
- Receive the execution trace, nonce and the encrypted measurement of the software stack from the service provider for a specific execution of the web service.
- Decrypt the data in the TPM using the private part of it's non-migratable AIK to get the TPM measurement of the service provider's software stack upto the *requirements monitor* and verify the value for conformance.
- Check the nonce for freshness to guard against replay attacks.
- With the reference measurement as the basis, programmatically check the trace for violation of specific properties.
- If any of these tests fail, notify the user of a possibility of violation.
- If all the verifications were successful, send an assurance to the client.

For every transaction involving sensitive data, the client initiates the monitoring procedures on the service provider's system as well as on the trusted third party. As the transaction proceeds, the *requirements monitor* monitors the integrity of the service implementations. At the end of the transaction, the client gets either an assurance or a violation alert from the trusted third party.

The next chapter summarizes the results of evaluating this architecture on the subject web services. It is demonstrated that integrity verification of web services can be achieved with a very short time-overhead if Aspect monitors are used.

CHAPTER 6. Evaluation

The goal of this chapter is to validate the following two claims:

1. **Feasibility**: The proposed extension of SOA can be implemented in a realistic setting.

2. **Effectiveness**: The proposed architecture is useful for monitoring non-functional requirements of web services.

To validate the first claim of feasibility, the details of a working implementation of the proposed architectural extension was provided in the Chapter 5. To support the second claim, popular web services were selected and monitored for integrity. The subjects for our case study were selected from the web service implementations available from the Apache Axis distribution. Table 6.1 briefly describes the web services used for this case study and the sections of the service implementation that was traced by the requirement monitor for each web service. Some of these sections were chosen randomly while others were chosen to monitor certain methods handling specific data, labeled as sensitive. The Section 6.3 presents a detailed analysis of the overheads incurred in monitoring the web services for their corresponding non-functional properties. It is also shown that the time lag caused by monitoring is negligible and thus, the proposed architecture effectively monitors the integrity of the web services.

6.1 Experimental Setup

The experimental setup was implemented using two Dell Precision 390 stations each having Intel Core2 Duo Processor @ 1.86 GHz and 2 GB of RAM. The processors on these stations have a TPM (Version 1.2) manufactured by Atmel Corporation, embedded in them with 24 PCRs each. One of the stations is assigned the role of a service provider while the other plays

Table 6.1 Subjects for our Case Study

Service name	**Short description**	**Traced Sections of the Service Implementation**
Stock	Gets quote for the stock ”symbol”	1. Instructions invoking setters. 2. Methods with private access.
Echo	Echoes a string	1. Method entries and exits. 2. Methods with public access.
Encoding	Serialization of a message	1. Methods with private access. 2. Instructions which invoke getters.
Message	Simple XML messaging service	1. Methods with private access. 2. Instructions invoking getters.
Bidbuy	Request for a quote, purchase a given quantity of a specified product and process purchase order.	1. Method entries and exits. 2. Instructions that invoke getters and setters.

the role of a trusted third party. We used *tpm4java* for developing our trust analyzer to take integrity measurements of the requirements monitor on the service provider's side. The Java library *tpm4java*, developed by Tews et al. [43], is used for accessing the TPM functionality from Java applications. The test environment consists of Apache Web server Version 2.2, Tomcat Servlet Container Version 5.5.23 and Axis SOAP server Version 1.2 running on Windows XP Professional operating system. For evaluating the requirements of the web service implementation, we use a commercial software called $CodeMonitor^{TM}$ (monitor) from Tangentum Technologies [14] as our subject monitor. The monitor instruments the Java bytecode to log certain actions and this makes it possible to monitor web services that have already been deployed. For doing these, it must be installed in the same environment as that of the web service. For the purpose of this experiment, we defined the requirement as, *” The execution trace of the program involving the variables and methods dealing with client data labeled as sensitive, should not include APIs dealing with persistence or serialization.”*

6.2 Violations

This class of compromise involving only the violation of requirements can be detected by current approaches for requirements monitoring (e.g. [26, 35]). Using similar techniques, our subject monitor was also able to give the execution trace for methods that caused either persistence or serialization of data. When such a violation occurs, the trusted third party can signal the end user of a breach of trust by the web service.

However, current approaches do not detect violation when the requirement monitor is itself compromised. Since the monitor has to be installed in the service provider's environment, the monitor can be compromised in many ways. For this paper, we instrumented the monitor to report a normal trace even when there was a violation of trust. Thus, the integrity of the web service is a function of the integrity of the requirements monitoring software. One such case is presented in Figure 7, in which one of the library files of the monitor is altered. Since the monitor itself is being monitored by the hardware-based TPM, it is possible to detect such a violation.

Table 6.2 TPM Measurements for a Genuine and a Compromised Requirements Monitor

Files Monitored by TPM	160-bit SHA1 Hash of Genuine Monitor	160-bit SHA1 of the Compromised Monitor
../codemonitor.license	6476...DB8F	6476...DB8F
	...	
../codemonitor.config	8D9E...5FA8	8D9E...5FA8
../codemonitor.jar	23F9...5BA2	D843...1531
../jbcs-client.jar	86AA...BE56	0F66...00F7
	...	

From Table 6.2, it can be observed that the hash values in the third column starting from the entry corresponding to the file *codemonitor.jar* differ significantly from their corresponding entries in the second column. This is because the SHA1 hashing algorithm in the TPM not only hashes the listed files but also preserves the order of hashing. It implies that at least one of the library files including *codemonitor.jar* has been altered without the knowledge of the trusted third party.

6.3 Overhead of Monitoring

Table 6.3 compares the average time taken to execute a web service in a standalone manner, when CodeMonitor is used and when custom Aspects are applied for monitoring the web service implementation for the properties listed in Table 6.1. These values are the averages of the time taken to execute the service over ten client requests. The overhead due to CodeMonitor is greater because, it instruments all the instructions used in the web service implementation including those of the libraries, at run time. Since the source code for CodeMonitor was not available, we could not circumvent this overhead. So, we wrote custom aspects to monitor the same sections of the service implementation and achieved a better performance. This validates the claim that web services can be monitored for integrity without a tangible time lag in responding to the client's request.

Table 6.3 Overhead of Monitoring

Service	**Execution Time without any monitor (in seconds)**	**Execution time with *CodeMonitor* (in seconds)**	**Execution time with Aspect Monitor**
Stock	0.944	10.688 11.476	1.283 1.005
Echo	1.299	42.375 12.812	1.609 1.640
Encoding	0.738	11.828 9.621	0.922 1.026
Message	0.945	7.200 20.641	1.209 1.208
Bidbuy	0.993	83.110 10.900	1.349 1.341

CHAPTER 7. Case Study

In order to explain the method adopted to evaluate every web service, we will use an example web service (Shopping cart) shown in Figure 7.1. This figure depicts a common case of violation of trust by a web service involved in financial transactions. This service was used in addition to the case study subjects from Apache Axis project.

7.1 Example Web Service - Shopping Cart

The shopping cart web service consumes the credit card number, the card validation code (cvc) and the purchase order, as the input from the customer. At the end of the transaction, the customer gets back the invoice number, which is the output of the web service. In this example, the customer is unaware of the fact that the web service provider has processed the customer's input for adversarial purposes and that it has stored his credit card number within its local database.

The web service implementation could have been certified to be compliant at the time of deployment, but later, it might have been reprogrammed by the service provider with a malicious intent. This violation of trust goes undetected. In essence, without much ado, the web service provider has extracted the customer's credit card number and other important data. Currently, there are no established strategies for detecting unsafe persistence of data or for detecting adversarial computation in a service provider's environment.

7.2 Experiment

We created shopping cart web-service described in the previous section. The web service is invoked from a web browser of another machine. The web service accepts the credit card

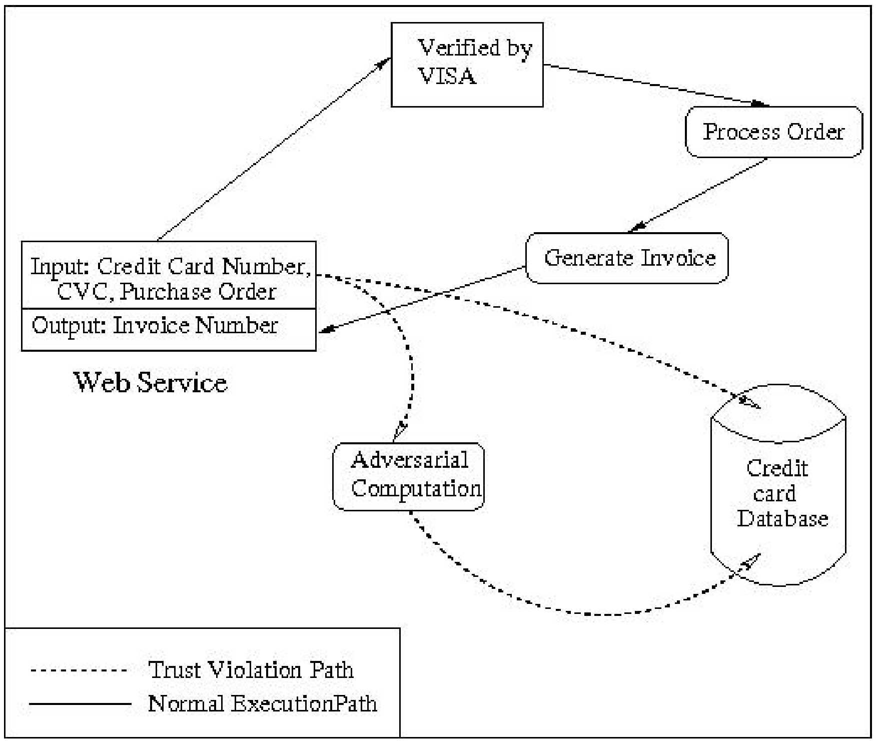

Figure 7.1 Example of Trust Violation

details and a list of items in the shopping cart as the input for the transaction and outputs the invoice, carrying out all the intermediate tasks from fulfilling the order to billing the client appropriately. We used the monitor to observe the definition and use of data fields carrying sensitive data. The trace produced by the monitor is shown in Figure 7.2. The monitor produces trace preceded by PUT and GET corresponding to the data fields carrying sensitive data.

```
ENTRY Connect.initialize()
PUT Connect.name="John Doe"
...
GET Connect.name "John Doe"
...
Order successfully processed: John Doe
```

Figure 7.2 Output trace for the original service

Since the monitor is also in the service provider's environment it can be compromised to not to report the violations. We mimicked the compromise by instrumenting the monitor to ignore any violations that might have occurred. Such a *requirements monitor* can be used by a service provider to report wellness when all is not well with the web service. This is when a hardware-based trust mechanism can falsify a wrong claim of the service provider. We used the trusted platform module to validate the *monitor* to ensure that it is not compromised or changed. Next subsections report on the following two cases of requirements violation:

1. The monitor detects a violation in the web service.

2. The monitor itself is compromised and hence the violations are not reported. However, such a compromise is detected by the hardware-based trust mechanism.

7.2.1 Monitor detects trust violation

This class of compromise can be detected by current approaches for requirements monitoring (e.g. [16, 17, 26, 35]). Using similar techniques, our prototype requirements monitor was also able to detect the compromise.

```
ENTRY com.mysql.jdbc.Statement.executeQuery
("SHOW VARIABLES")
ENTRY com.mysql.jdbc.Connection.prepare
Statement
("INSERT INTO cnumbers(name,ccn,cvc) VALUES
(?,?,?)")
...
```

Figure 7.3 Trace for compromised web service

Figure 7.2 shows the trace generated for the clean-room web service and the Figure 7.3 shows the execution trace of the compromised web service. Here, storing the credit card number in the database is considered a violation of integrity by the web service. Since the authentication server on the trusted third party stores the execution trace of the clean-room implementation, it can detect any violations by comparing the latest trace with that of the original one. When such a violation occurs, the trusted third party can signal the end user of a breach of trust by the web service.

7.2.2 Detecting a compromised monitor

Since the monitor has to be installed in the service provider's environment, the monitor itself can be compromised without much difficulty in a realistic setting. For this paper, we have instrumented the monitor to compromise its nature. There are many other ways of compromising the monitor itself. If the monitor can be decompiled, the source code can be changed to suit one's interests. Another way can be to instrument the monitor to ignore the violations. When this is done, the *requirements monitor* reports the trace as shown in figure 7.2 even when there is a violation of trust. Thus, the integrity of the web service is a function of the integrity of the requirements monitoring software. One such case is presented in Table 7.1 in which one of the library files of the monitor is altered. Since the *monitor* itself is being monitored by the hardware-based TPM, it is possible to detect such a violation. A service provider can also replay the same trace after every transaction. To prevent such occurrences, the trust analyzer on the service provider's side requests the trusted third party for a nonce which will be used by the service provider's TPM to compute the hash of the *requirements monitor*.The *requirements monitor* should also include this nonce in every trace it generates to prove that the measurements are fresh.

Table 7.1 Case Study: Comparison of TPM Measurements between a genuine and a compromised Monitor

File	160-bit SHA1 Hash of Genuine Monitor	160-bit SHA1 of the Modified Monitor
../readme.txt	5887...2111	5887...2111
../codemonitor.license	6476...DB8F	6476...DB8F
../Connect.class	9F2B...A638	9F2B...A638
../Connect.java	1394...BC86	1394...BC86
../codemonitor.config	8D9E...5FA8	8D9E...5FA8
../codemonitor.jar	23F9...5BA2	D843...1531
../jbcs-client.jar	86AA...BE56	0F66...00F7
../jdax.jar	85F7...30E2	2020...8D15
../xjbc-jdax.jar	944E...1F3E	99CA...547F

The first column of the Table 7.1 lists the files of monitor being monitored by the TPM. The second column shows the 160-bit hash values of PCR #10 during the clean-room measurement of the software. The third column shows those measurements that were obtained after the class files of one of the library files were altered. It can be observed that the hash values in the third column starting from the entry corresponding to the file *codemonitor.jar* differ significantly from their corresponding entries in the second column. This is because the SHA1 hashing algorithm in the TPM not only hashes the contents of the candidate files but also preserves the order in which the files were hashed. This implies that at least one of the library files including *codemonitor.jar* has been altered without the knowledge of the trusted third party. The list of files that were monitored, which includes property files, system files and executables, is long and only a subset of this list is published in this paper to demonstrate the viability of the concept.

This case study demonstrated that the proposed architecture can not only detect the integrity violation by the web service but also detect the tampering of the *requirements monitor* software and can produce evidence for both events.

CHAPTER 8. Conclusions

The goal of this thesis was to evaluate the web services for non-functional requirements or concerns such as privacy and integrity. The traditional service oriented architecture lacks a layer or an interface that is capable of inspiring trust in the clients. To evaluate a web service implementation for integrity, there has to be a monitoring software to oversee the execution of a web service implementation and report accurate data regarding this execution. Such a monitor has to be deployed in the remote environment. To trust the results sent by the monitor, there must be a mechanism of validating monitors deployed in the remote environments. However, we showed that it is very difficult to trust a monitor deployed in a remote or untrustworthy environment using any of the software-based approaches.

We proposed an extension to the traditional service oriented architecture to incorporate a layer of *trust negotiation and verification* other than the usual service negotiation. The *trust negotiation* interface helps to establish a contract of trust between the service provider and a trusted third party. This contract is verified by the *trust verification* interface by securely communicating with the Trusted Platform Module (TPM) on the service provider's environment. The root of trust in our architecture is derived from the hardware (TPM) rather than software. Thus, the proposed architecture makes it possible to give an assurance to the client with a high level of confidence.

The implementation of the proposed architecture was evaluated using the standard web services available with the Apache Axis Distribution. The evaluation demonstrated the feasibility of implementing the proposed extensions to the SOA in a realistic setting. It also demonstrated that the proposed architecture was effective in monitoring the non-functional requirements of the web services.

8.1 Discussion

Existing security models for web-services mostly consider four security and trust issues in the service infrastructures. The first question is whether the requesting entity, i.e. client, is who they claim to be. The second question is whether the client is authorized to use the service. The third question is whether the data, i.e. service request and reply messages, exchanged between the client and the service provider is protected from unauthorized access and from tampering. The fourth question is whether the client and/or the provider are protected from the each other's denial of service attacks.

These questions, while important, do not address a key concern of clients. Provided a reasonable security framework is available, the client gets the guarantee that the service request and replies will be protected from unauthorized access and tampering; however, these frameworks do not offer any guarantee whether the data will remain private and tamper-proof in the application domain of the service provider. Note that the application domain of the service provider is where the client's service requests are processed and replies returned. A service oriented architecture is only as secure as its weakest link. In a truly decoupled environment, which is the main motto of SOAs, including constructs to negotiate, enforce, and verify trust and security guarantees within the provider's application domain through the service discovery interfaces thus seems to be a crucial pre-condition of mission-critical deployment of SOAs.

My work thus creates a timely opportunity to rethink the traditional notion of trust in service oriented architecture.

8.2 Future Work

Contracts are important for web services. A framework that lets the client specify certain properties that are to be enforced on the web service implementation will make this architecture complete. Behavioral interface specification languages like Java Modeling Language [11] are widely used to specify behavior of Java modules. It is a powerful Design by Contract (DBC) tool for Java. Can a specification language similar to JML be designed to enforce contracts such as data privacy and integrity on the remote web service implementations?

It would be very helpful if there are customized virtual machines available that are capable of enforcing integrity requirements on the executable code. This leads us to another interesting direction. Can certain modules which can enforce specific trust properties at the byte code level of web service implementations be incorporated in a Virtual Machine?

There is a potential overhead in the monitoring process. The entire source code is monitored for deviations. For small implementations, there is hardly any overhead. However, as the size of the implementation increases, there would be a tangible time delay caused by such verification mechanism. Can we use program analysis techniques such as data-flow graphs or control-flow graphs to restrict the scope of monitoring to segments of code that deal with sensitive data?

One of the primary goals of security architectures is to prevent data leaks from a more secure/sensitive level to the lower levels. Information flow mechanisms involve building architectures to prevent such insecure data flows. Is it possible to build an architecture to ensure secure information flow in the web service deployed on this architecture?

This research work explored the aspect of finding mechanisms for enforcing non-functional requirements and security properties in web services. The above mentioned areas can be explored further in the context of Service Oriented Architecture along the lines of this research work. Any breakthrough in that direction will contribute towards building an efficient, trustworthy and secure Service Oriented Architecture.

Bibliography

[1] Embedded security.
http://www.atmel.com/products/Embedded/.

[2] Tpm main part 1 design principles specification version 1.2.
https://www.trustedcomputinggroup.org/specs/TPM.

[3] R. Anderson. Cryptography and competition policy: issues with 'trusted computing'. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 3–10, New York, NY, USA, 2003. ACM Press.

[4] S. Anderson and et al. Web services trust language (wstrust).
http://msdn.microsoft.com/ws/2004/04/ws-trust/.

[5] W. A. Arbaugh, D. J. farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *IEEE Symp. Security and Privacy*, pages 65–71, IEEE CS Press, Los Alamitos, California, 1997.

[6] S. Bajikar. Trusted platform module (tpm) based security on notebook pcs - white paper. Technical report, Mobile Platforms Group Intel Corporation, June 2002.

[7] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS '06*, pages 63–71.

[8] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *ICSOC '04*, pages 193–202.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system.

In *15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, 1995.

[10] G. Canfora and M. D. Penta. Testing services and service-centric systems: Challenges and opportunities. *IT Professional*, 8(2):10–17, 2006.

[11] Y. Cheon and G. Leavens. A runtime assertion checker for the java modeling language. In *Software Engineering Research and Practice(SERP'02), CSREA Press, June 2002*, pages 322–328, 2002.

[12] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. Technical report, World Wide Web Consortium, March 2001.

[13] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *1987 IEEE Symposium on Security and Privacy*, pages 184–194, 1987.

[14] *codemonitor*^{TM}.
http://www.tangentum.biz/.

[15] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[16] M. S. Feather, S. Fickas, A. V. Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *IWSSD '98*, page 50.

[17] S. Fickas and M. S. Feather. Requirements monitoring in dynamic environments. In *RE '95*, page 140.

[18] V. Haldar and M. Franz. Symmetric behavior-based trust: a new paradigm for internet computing. In *NSPW '04: Proceedings of the 2004 workshop on New security paradigms*, pages 79–84, New York, NY, USA, 2004. ACM Press.

[19] M. Hosamani, H. Narayanappa, and H. Rajan. How to trust web services monitor executing in an untrusted environment? In *3rd International Conference on Next Generation*

Web Services Practices, page To appear, Washington, DC, Oct 2007. IEEE Computer Society.

[20] M. Hosamani, H. Narayanappa, and H. Rajan. Monitoring the monitor: An approach towards trustworthiness in service oriented architecture. In *2nd International Workshop on Service Oriented Software Engineering (IW-SOSWE 2007)*, September 2007.

[21] M. Hosamani, H. Narayanappa, and H. Rajan. Monitoring the monitor: An approach towards trustworthiness in service oriented architecture. Technical Report 07-07, Iowa State University, Ames, Iowa, June 2007.

[22] G. Jayaraman, V. P. Ranganath, and J. Hatcliff. Kaveri: Delivering indus java program slicer. In *Fundamental Approaches to Software Engineering, FASE 2005, Springer-Verlag*, April 2005.

[23] C. Kaler and et al. Web services security (ws-security). http://msdn.microsoft.com/library/enus/dnglobspec/html/ws-security.asp.

[24] Y. Katsuno, Y. Watanabe, S. Yoshihama, T. Mishina, and M. Kudoh. Layering negotiations for flexible attestation. In *First ACM workshop on Scalable Trusted Computing*, November 2006.

[25] D. Kuo, A. Fekete, P. Greenfield, S. Nepal, J. Zic, S. Parastatidis, and J. Webber. Expressing and reasoning about service contracts in service-oriented computing. In *ICWS '06*, pages 915–918.

[26] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Monitoring and control in scenario-based requirements analysis. In *ICSE '05*, pages 382–391.

[27] K. Mahbub and G. Spanoudakis. Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. In *ICWS '05*, pages 257–265.

[28] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan. 1997.

[29] Microsoft next-generation secure computing base.
www.microsoft.com/resources/ngscb.

[30] M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE '03*, page 3.

[31] M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing: Introduction. *Commun. ACM*, 46(10):24–28, 2003.

[32] S. Park, L. Liu, C. Pu, M. Srivatsa, and J. Zhang. Resilient trust management for web service integration. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, pages 499–506, Washington, DC, USA, 2005. IEEE Computer Society.

[33] B. Pfitzmann, J. Riordan, C. Stuble, M. Waidner, and A. Weber. The perseus architecture. Technical Report RZ3335 (#93381), IBM Research Division, April 2001.

[34] A. Rezgui, M. Ouzzani, A. Bouguettaya, and B. Medjahed. Preserving privacy in web services. In *WIDM '02*, pages 56–62.

[35] W. Robinson. Monitoring software requirements using instrumented code. In *HICSS '02*, page 276.2.

[36] A. Sabelfeld and A. Myers. Language-based information-flow security. In *IEEE Journal on Selected Areas in Communications, 21(1), 2003.*, 2003.

[37] A. Sadeghi and C. Stuble. Taming trusted platforms by operating system design. In *Fourth International Workshop, Information Security Applications*, volume 2908 of *Lecture Notes in Computer Science (LNCS)*, 2003.

[38] R. Sailer, L. van Doorn, and J. P. Ward. The role of TPM in enterprise security. Technical report, IBM Research, October 2004.

[39] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Thirteenth Usenix Security Symposium*, pages 223–238, August 2004.

[40] M. Schroeder. Engineering a security kernel for multics. In *Fifth Symposium on Operating Systems Principles*, pages 125–132, November 1975.

[41] H. Skogsrud, B. Benatallah, F. Casati, and F. Toumani. Managing impacts of security protocol changes in service-oriented applications. In *2007 IEEE International Conference on Software Engineering*, 2007.

[42] Trusted computing group.
https://www.trustedcomputinggroup.org.

[43] E. Tews and M. Hermanowski. Projektvorstellung tpm4java trusted computing fur java.
http://tpm4java.datenzone.de.

[44] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO 37*, pages 243–254, 2004.

[45] H. Wada, J. Suzuki, and K. Oba. Modeling non-functional aspects in service oriented architecture. In *IEEE International Conference on Services Computing (SCC'06)*, pages 222–229, 2006.

[46] X. Wang, Y. L. Yin, and H. Yu. Collision search attacks on sha1.
http://www.cryptome.org/sha-attacks.htm.

[47] S. Yoshihama, T. Ebringer, M. Nakamura, S. Munetoh, and H. Maruyama. Ws-attestation: Efficient and fine-grained remote attestation on web services. In *IEEE International Conference on Web Services (ICWS'05)*, July 2005.