2008

Tisa: Toward Trustworthy Services in a Service-Oriented Architecture

Hridesh Rajan
Iowa State University, hridesh@iastate.edu

Mahantesh Hosamani
Iowa State University

Tisa: Toward Trustworthy Services in a Service-Oriented Architecture

Abstract
Verifying whether a service implementation is conforming to its service-level agreements is important to inspire confidence in services in a service-oriented architecture (SoA). Functional agreements can be checked by observing the published interface of the service, but other agreements that are more non-functional in nature, are often verified by deploying a monitor that observes the execution of the service implementation. A problem is that such a monitor must execute in an untrusted environment. Thus, integrity of the results reported by such a monitor crucially depends on its integrity. We contribute an extension of the traditional SoA, based on hardware-based root of trust, that allows clients, brokers and providers to negotiate and validate the integrity of a requirements monitor executing in an untrusted environment. We make two basic claims: first, that it is feasible to realize our approach using existing hardware and software solutions, and second, that integrity verification can be done at a relatively small overhead. To evaluate feasibility, we have realized our approach using current software and hardware solutions. To measure overhead, we have conducted a case study using a collection of Web service implementations available with Apache Axis implementation.

Keywords
Validation, Assertion checkers, assertion languages, performance, Verification, Monitors, Domain-specific architectures

Comments
This article is published as Rajan, Hridesh, and Mahantesh Hosamani. "Tisa: Toward trustworthy services in a service-oriented architecture." IEEE Transactions on Services Computing 1, no. 4 (2008): 201-213. 10.1145/2858965.2814289. Posted with permission

Tisa: Towards Trustworthy Services in a Service-oriented Architecture

Hridesh Rajan Mahantesh Hosamani
Dept. of Computer Science, Iowa State University
E-mail: {hridesh, mahantesh}@cs.iastate.edu

Abstract

Verifying whether a service implementation is conforming to its service-level agreements is important to inspire confidence in services in a service-oriented architecture. A part of these agreements, in particular those that are functional in nature, can be checked by observing the published interface of the service, but other agreements that are more non-functional in nature, are often verified by deploying a monitor that observes the execution of the service implementation. A key problem is that such a monitor must execute in an untrusted environment (at the service provider's site). Thus, integrity of the results reported by such a monitor crucially depends on its integrity. The key technical contribution of this article is an extension of the traditional notion of a service-oriented architecture that allows clients, brokers and providers to negotiate and validate the integrity of a requirements monitor. We describe an approach, based on hardware-based root of trust, for verifying the integrity of a requirements monitor executing in an untrusted environment. We make two basic claims: first, that it is feasible to realize our approach using existing hardware and software solutions, and second, that integrity verification can be done at a relatively small overhead. To evaluate our feasibility claim, we present a realization of our approach using a commercial requirements monitor. To measure overhead, we have conducted a case study using a collection of web service implementations available with Apache Axis implementation.

I. Introduction

Service-Oriented Computing (SoC) is a new programming paradigm that promotes dealing with modules or components, often called *services* [1]. These modules could be represented as reusable entities with specific goals or objectives. Furthermore these entities can be composed to form larger applications or solutions [1]. Among other goals, service-oriented computing promotes abstraction, loose coupling and interoperability among services [2]. This is often achieved by introducing a published interface (often a description written in an XML-based language such as WSDL [3]), for communication between services and clients [2]. This interface is used by three types of entities: providers, brokers, and clients, for communication. The exchange follows the sequence publish-find-bind-execute to discover and use services [2]. By allowing components to be decoupled using a specified interface, service-oriented computing enables platform-independent integration. These new integration possibilities are valuable for constructing today's interoperable, large-scale, complex software-intensive systems.

The published interface of a service describes the functional requirements for co-ordination between service implementations and clients. For example, the published interface for a credit card processing service may expect clients to provide the transaction details and expect the service implementation to produce a confirmation number. The specification of input (transaction details) and output (confirmation number) describe the functional requirements for this service. Verifying whether a service (or its composition) satisfies its requirements is an important problem [4], [5], [6], [7].

To verify whether a service is satisfying its functional requirements it suffices to observe or test the published interface [4], [5], [6], [7]. However, to validate a non-functional requirement such as "R2: the service shall not persist the credit card number supplied by the client", it may not be sufficient to validate just the external interface. Validation of such requirements may only come from a monitor (such as those described in [8], [9], [10], [11]) 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.

Service providers may deploy such monitors and make the monitor's functionality available to their clients, possibly using an XML-based interface. However, presence of such monitor itself is insufficient. This is primarily because services are often deployed on servers that are not owned or operated by the clients. Even if the client is willing to trust the results of the monitor, there is no guarantee that the monitor itself has not been compromised.

An important contribution of this article is the identification of the need for validating the integrity of the monitor operating at the service providers site. To illustrate the need, let s ($\in S$) be a service specification, i ($\in I$) be a service implementation, $M : S \times I \rightarrow \{true, false\}$ be a monitor that is capable of detecting deviations in the execution of the service implementation from its specification running in a trusted environment, and $M' : S \times I \rightarrow \{true, false\}$ be a monitor that is similarly capable, but may be running in an untrusted environment. The problem is to validate whether $M \equiv M'$.

Let us assume that a validation mechanism $V' : M \times M' \rightarrow \{true, false\}$ exists. We argue that part of V', however small that may be, must run in the same untrusted environment to observe M' so that it can be compared with M. If not, V' will depend on the untrusted 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 untrusted 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 without an adequate mechanism in which trust can be placed by all involved entities.

The second important contribution of this work is the insight that a hardware-based mechanism can be used as a root of trust in the presence of distributed services in a service-oriented computing environment. The main intuition is that if the hardware-based root of trust ensures that there exists a $\delta V'$ such that we do not need another monitor to verify its integrity, $\delta V'$ would make V' realizable. Recent research results on *Trusted Platform Module* (TPM) [12], [13] make realization of such hardware-based root of trust feasible. TPM is a co-processor that is now being shipped with every CPU of major processor vendors such as Intel and AMD and is therefore available broadly.

The technical underpinnings of this work include an architectural extension and associated verification algorithms[1] that utilize TPM and check whether the monitor executing on the service provider's site is not corrupted. Formally using the terminology above, it validates whether $M \equiv M'$. Together these make trusted service-oriented architectures realizable. In the next subsection, we define the scope of our approach.

[1]Initial ideas were proposed in our prior work [14], [15].

A. Scope of This Article

This section broadly relates our work to other related ideas to precisely characterize our contributions. In other words, following should be read as "what we do not claim". It can also be safely skipped on first reading.

Requirements monitoring. This article does not propose an approach for runtime requirements monitoring, there are many other research papers on this topic e.g. [8], [9], [10], [11]. To simplify the discussion and our experimental setup, we have used a trace-based requirements monitor (see Section IV), however, our ideas are applicable to other types of requirement monitors as well.

Functional Requirements. Our work does not propose a heavyweight requirements monitoring for validating functional requirements as they can very well be monitored by verifying the externally visible interface of the service as shown by others [4], [5], [6], [7].

Notion of Integrity. We have intentionally not restricted ourselves to a specific notion of integrity in this article. Any existing notions, along with a corresponding verification mechanism can be used. In the examples presented in this article, we have used a notion based on checksum. Briefly, in these examples we consider that a monitor's integrity has not been violated, if its checksum as computed by the trust analyzer and signed by the TPM matches the cleanroom measurements. Our approach can be adapted to use more sophisticated models based on functional equivalence, however, for the proof of concept we consider checksum-based notion of integrity to be sufficient.

Secrecy and Authenticity Issues. Our work is orthogonal and complementary to the secrecy and authenticity research in the web services security community. We do not focus on securing the interaction between service providers, brokers and clients, which has been the main focus of many existing approaches, e.g. current standards such as WS-Security [16] and WS-Trust [17] or proposals such as that by Skogsurd et al. [18].

These approaches address the issue of security-token interoperability and secure transactions. They do not address the integrity issues for components services and they cannot be used directly to certify indisputable trust in an untrusted environment. Our approach builds upon existing work on secrecy and authenticity to develop a mechanism for trusting loosely-coupled components in a service-oriented computing environment.

The rest of this article is organized as follows. Section II describes trusted platform modules, which form the basis of our proposed architecture. Section III describes key ideas of this work. Section IV-V evaluate these contri-

butions. In particular, the former evaluates the feasibility claim and the later evaluates the utility claims. Section VI compares and contrasts our work with related approaches. Section VII discusses potential adoption paths for our work in the current service-oriented computing research and practice. Section VIII discusses future work and concludes. We now describe key parts of the trust platform module.

II. Background: Trusted Platform Modules

Over the past few years, a consortium of key industry players under the umbrella of Trusted Computing Group (TCG) [19], have developed the specification for the trusted platform module (TPM) [20] with the goal to guarantee security, integrity and confidentiality of data through innovative hardware-based architectures. Both TCG and alternatives [21], aim to bootstrap higher level trust from rudimentary TPM supported trust using some software trust architecture or design principle.

A 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 [19]. 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 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 upper layers. In what follows we describe its key components.

A. Cryptographic Coprocessor

The cryptographic coprocessor implements cryptographic functions executed within the TPM hardware. Hardware or software entities outside the TPM have no 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 measurements during the platform boot phases.

B. Random Number Generator (RNG)

A RNG is the source of randomness in TPM. It is provided for key generation, nonce generation and for randomness in signatures. This capability is protected from external access.

C. Platform Configuration Registers (PCRs)

PCRs are set of registers that 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. Thus 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.

D. 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 their 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.

E. Attestation Identity Keys (AIKs)

AIKs are used by a *privacy certification authority* to present different keys to different remote parties to enable the system to hide its platform identity from other systems.

F. Certificates

The TPM is also equipped with three kinds of certificates [22]: endorsement, platform, and conformance. An *endorsement certificate* attests that a particular platform configuration is genuine. This contains the public part of the endorsement key. The *platform certificate* attests that the security components of the platform are genuine. This is provided by the platform vendor, and the *conformance certificate* can be provided by a third party to certify the security properties of the platform.

G. TPM Usage Models

Bajikar [22] describe three usage models of the TPM. First, *hardware protected storage*, where TPM is 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. Second, *information binding*, where critical data is bound

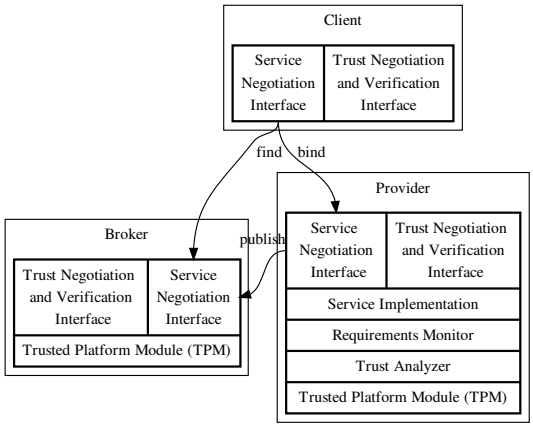

Fig. 1. Proposed Architectural Extensions

to a platform such that it is accessible only if the conditions specified during the binding are met and rendered inaccessible if migrated to a different platform, and third *platform authentication*, where attestation identity keys are always bound to the platform. These can be used to authenticate the user and the platform. Our technique uses the third model to authenticate the service implementation platform, including the requirements monitor.

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 (c.f. [23]). Alternatives such as PERSEUS are also proposed [24], [25]. Although the Jury is still out on the social aspects of TPMs, their wide availability and advantages combine to warrant research on the use of these mechanisms for trusted service-oriented architectures.

III. Monitoring the Monitor

A. Overview

Our approach consists of architectural extensions and algorithms for verifying integrity of a remotely- hosted requirements monitor. In a spirit similar to the published functional interface for service negotiation, entities in a service-oriented architecture are extended to publish a trust-negotiation and verification interface. New components are added to support the trust-negotiation interface. A trusted platform moduler (TPM) and trust analyzer are added to the service provider and a TPM is added to the broker. The trust analyzer verifies the integrity of the requirements monitor executing in the domain of the service provider. The TPM on the service provider's site is used to attest to the integrity of the provider's software stack as well its identity. The TPM on the service broker's site is used to attest to the identity of the service broker. The broker negotiates desired integrity-level with the provider and provides assurance to the client using the trust-negotiation and verification interface.

Traditional interaction pattern *publish-find-bind* between components in SoAs is augmented with steps for negotiating requirements monitoring capabilities as shown in Figure 2. During the publish step, in addition to the functional properties service providers also make the capabilities of the requirements monitor available to the broker via the trust negotiation interface. By capabilities, we mean the type of non-functional requirements that the monitor can verify. For example, a service provider that deploys a monitor similar to Barbon *et al.* [5]'s work would make the capabilities to verify Boolean, statistical, and timing properties available.

During the find step, a client sends a request to the broker for services that satisfy desired functional properties as well as provide monitoring capabilities to check desired non-functional properties. The broker responds to this request with a list of only those services that satisfy both functional requirements as well as provide monitoring capabilities for non-functional requirements. During the bind step, in addition to sending a service request to the service provider, a client also sends an attestation request to the service broker. The broker attests to the fact that for that specific transaction the non-functional properties claimed by the provider indeed hold. The notification from broker as well as service together complete the bind step. The TPM plays a vital role in the publish and bind step.

The rest of this section describes the architectural extensions and corresponding algorithms.

B. Architectural Extensions

Our architectural extensions are shown in Figure 1. The key addition is a new interface, which we call *trust negotiation and verification interface*, between service provider, client, and broker. Those well-versed with networking terminologies can think of communication that take place using this interface as "out-of-band" compared to sequence of protocols (typically find-bind-execute) that takes place on the regular channel. The purpose of this interface is to facilitate negotiation of the desired level of trust between the components of the SOA.

In our extension, traditional component service broker also plays the role of a *trusted third party*. The goal of the trusted third party is to facilitate trust negotiation between clients and service providers, detect non-compliance of service providers, and to notify clients.

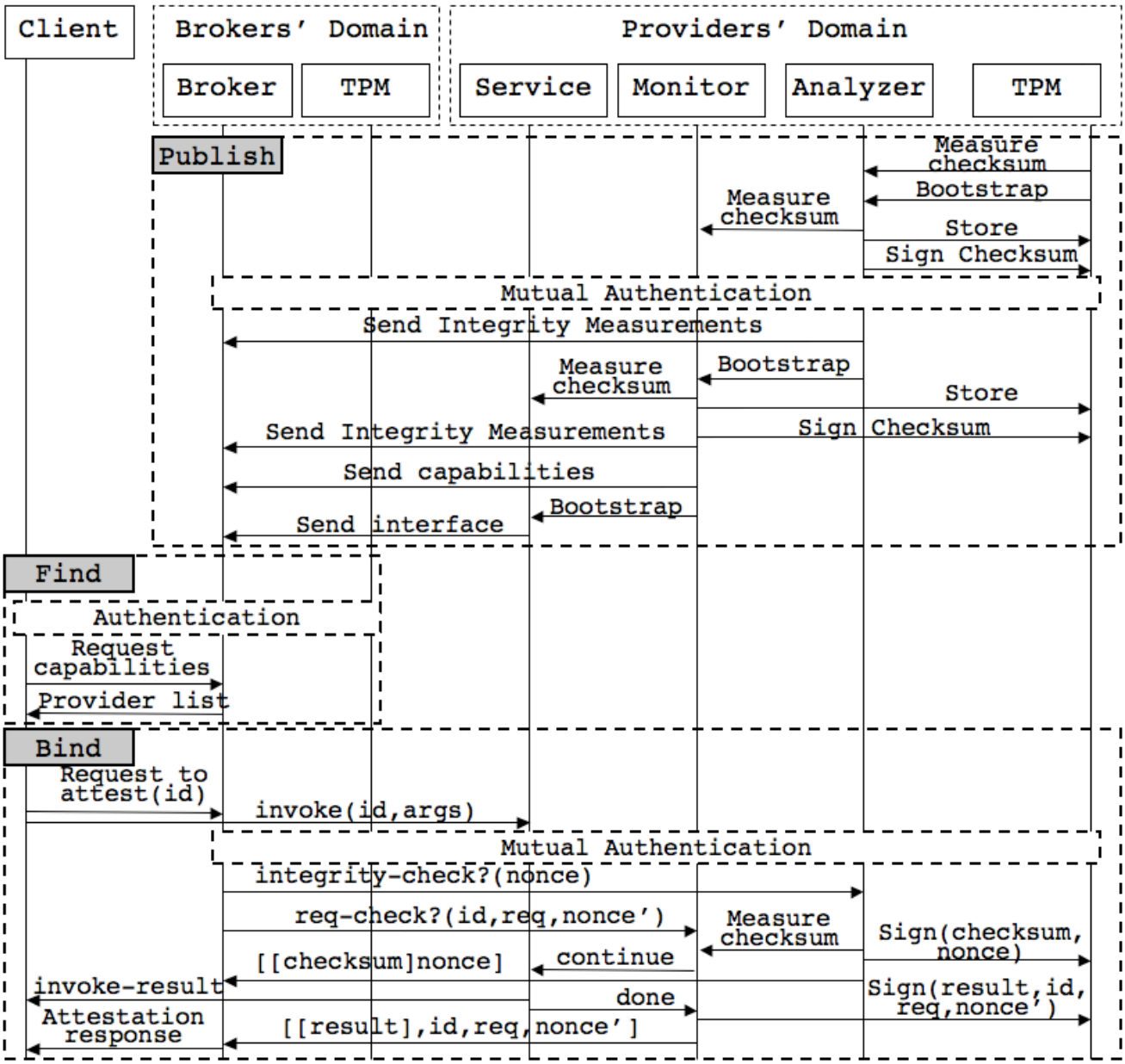

Fig. 2. Overview of Interaction between Components

New components on the brokers' side include a TPM and on the service providers' side includes a requirements monitor, a trust analyzer, and a TPM. No new components are needed on the client's side (although if one decides to have a symmetric notion of trust, a TPM on the client-side would be able to facilitate that). The role of these new components are as follows. The requirements monitor on the service provider's side verifies conformance of the service implementation to the desired non-functional requirements. The trust analyzer and the TPM on the service provider together attest to the validity of the monitor. The key insight that allows this combination to attest to the validity of the monitor is that the integrity measurements stored inside a TPM in a local environment cannot be changed, even by the owner of the system. Such a measurement can be read by anyone, though. Finally the TPM on the brokers' side is used to attest to the identity of the broker.

C. Interaction between Broker and Client

A client and a broker make use of the trust negotiation interface on two occasions. During the find step, in addition to the functional requirement, a client will also provide the trust and data integrity requirements. For example, for the payment processing service discussed in Section I, in addition to the desired functional requirement, the client may also specify that the service must not persist the argument credit card number. During this step, the TPM on the service broker's side is used to sign the response of the broker to the client. This assures the client that results of find are received from a genuine broker.

During the bind step, a client also communicates with the broker to request attestation for a service request. By attestation we mean that the broker verifies whether the service provider satisfied the previously agreed upon non-functional requirements during the service request.

D. Interaction between Provider and Broker

First step in publishing a service is to bootstrap the service implementation in a trustworthy manner. To achieve that a small bootstrapping mechanism similar to Sailer *et al.*'s TCG based Integrity Measurement Architecture for Linux [12] along with the TPM on the service providers' end is used to measure the integrity of the trust analyzer (Analyzer in Figure 2). This step is show as an arrow from TPM to Analyzer in Figure 2.

The small bootstrapping component used to measure the checksum of the Analyzer is not shown in the figure. The component essentially compares the current checksum of the Analyzer with the measurements in the cleanroom state and bootstraps the trust analyzer, if the current measurements are the same as cleanroom measurement. The trusted analyzer then bootstraps the requirements monitor (Monitor in Figure 2), which in turn starts the execution of the service implementation (Service in Figure 2) in a similar manner checking for integrity in each step.

During publish step, trust analyzer and requirments monitor send integrity measurements to the broker after mutual authentication. The integrity measurements are stored in the platform configuration registers (PCRs) of the TPM using SHA1 hashing algorithm. The key idea is to store integrity measurements as successive hash values. For example, let us assume that integrity measurements of trust analyzer resulted in a hash value of $hash_{TA}$ in the PCR. The integrity measurement for the requirements monitor now uses $hash_{TA}$ as its base value.

The SHA1 hashing algorithm is very effective for detecting tampering. 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 [26], thus collisions are very rare.

Before sending the integrity measurements to the provider, the trust analyzer uses the TPM to sign this measurements. The TPM's attestation identity key (AIK) is used to sign the PCR hash. AIK is a special purpose asymmetric signature key created by the TPM owner. The signing key of AIK is non-migratable and protected by the TPM. Since the signing key of the TPM's AIK cannot be retrieved by any user, this provides a proof that the data signed by this key was generated on the platform with the key. This provides platform authentication.

The cryptographic capabilities of the TPMs are leveraged to establish a secure communication channel between the service provider and the broker. This channel is used to convey the integrity measurements to the broker. Standard cryptographic mechanisms such as nonce are applied to prevent against attacks such as replay attacks, man-in-the-middle attacks, etc. A nonce is a random number that is used only once. It is included in all interactions of a particular session to prove the freshness of data. These are abbreviated as mutual authentication in Figure 2.

While publishing a service description (interface) with the broker the service provider also provides capabilities of the requirements monitor deployed in its environment. These capabilities represent the class of non-functional requirements that can be monitored during the service execution. The properties that can be monitored depends on the capabilities of the requirements monitor available at the service provider's location. In this work we focus on trace-based requirements monitor, but our notion is not limited to these.

In a trace-based monitor requirements are specified as logical formulas over program traces. An example of such trace-based requirement for the payment processing service could be, "in no trace of the service implementation, the argument credit card number directly or indirectly flows to a persistence-related method e.g. `File.write(...)`." Such properties can be expressed in standard logic such as Linear Temporal Logic (LTL) [27]. The detailed discussion of such logic is beyond the scope of this work. The provider would also facilitate a one time verification of the capabilities of the requirements monitor and implementation. We will discuss this in more detail in the context of our prototype.

During service execution, the broker and the provider must communicate to verify that: the provider and the broker are indeed the same principals that agreed to the contract during the publish step, the requirements monitor as well as the software stack on the provider's end is not compromised, and the trace provided by this monitor does not violate the properties desired by the client.

The trust analyzer component, which derives its root

of trust from the TPM on the provider's side, serves to verify the integrity of the requirements monitor. It provides a report that includes an immutable integrity measurement of the local architecture along with a time stamp to indicate the freshness of the measurement. The requirements monitor in turn determines whether trace of service execution satisfied desired non-functional requirements. TPM is also used to attest to the identity of the service provider.

The interaction between a client and a provider remains the same as in traditional service-oriented architectures.

IV. Evaluation: Feasibility

The goal of this section is to examine whether our proposed architecture can be realized using existing hardware platforms and software solutions. To that end, we describe the design and implementation of a prototype system that supports our proposed architecture. The rest of this section discusses various components starting with the experimental setup in the next section.

A. Experimental Setup

The hardware platform used for our prototype implementation was two Dell Precision 390 stations each with Intel Core2 Duo Processors running at 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 was assigned the role of a service provider while the other played the role of the broker. 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. [28], is also 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 used a commercial software called $CodeMonitor^{TM}$ (monitor) from Tangentum Technologies [29] 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."

For the purpose of this prototype implementation, we assumed that the operating system on the service provider's environment is secure, implying that the service provider will not be able to change the monitoring software without knowledge of the TPM in the system. A number of techniques such as the approach proposed by Sailer et al. to secure the operating system kernel can be used [13] to attain this goal. Interested readers are referred to this work for a detailed discussion of this issue.

B. Service Provider's Implementation

In our prototype, the service provider's implementation is augmented with requirements monitor and trust analyzer as shown in Figure 2.

Procedure 1 Publish step for providers' end

Input: Broker Address: brokerAddr, Service Id: Id

```
1: sendImpl(brokerAddr, Id);
2: sendMonitor(brokerAddr, Id);
3: sendConf(brokerAddr, Id);
4: progPoints = recvProgPoints(brokerAddr, Id);
```

The publish step on the service provider's side is shown in Procedure 1. The first step is to accept the requirements to be monitored. In our current prototype, requirements are expressed as fields and methods in the implementation that need to be monitored. We emulated the requirements identification process which consists of determining variables and methods dealing with data labeled as *sensitive*, using Kaveri [30], a tool for program slicing.

Program slicing is a technique for computing a subset of the program relevant to a property, often called the slicing criteria and expressed as either a value or a statement [31], [32], [33]. In future, we plan to have the requirement specifications (or slicing criteria) as a part of the web service interface itself, thereby making the process of requirements identification independent of the implementation of the web service. Our prototype can also be easily extended to support a full-fledged specification language such as the one we are currently developing [34], which allows policies specified in linear temporal logic [35] to be checked. However, we believe that the current workaround is sufficient to show the proof of concept.

Each transaction is validated by executing a series of steps shown in Procedure 2. First, the trust analyzer receives a nonce from the broker. Standard mechanisms for avoiding replay attacks and verifying the authenticity of the trusted third party are used here. A key part in this authentication session is played by the service provider's TPM, which signs the message containing the nonce with its key. During the computation of the results by the service

Procedure 2 Bind step for providers' end

Input: Broker Address: brokerAddr, Service Id: Id

1: nonce = recv(brokerAddr, Id);
2: startTraceCollection();
3: continueService();
4: trace = collectTrace();
5: checksum = computeSHA1Hash();
6: response = $sign_{AIK}(trace, nonce, checksum)$;
7: send(brokerAddr, Id, response);

implementation a trace is generated and maintained by the requirements monitor (steps 2-4).

This trace along with static checksum of the software stack is sent to the trusted third party (step 7). To remind the reader, the static checksum is created by computing the SHA1 hash of the software stack up to the requirements monitor and is essential to validate the current state of the service implementation (step 5). This hash is signed using the AIK on the TPM of the service provider (step 6). The message to the third party also contains the nonce originally obtained from the trusted third party to protect against replay attacks by the provider.

C. Broker's Implementation

In our prototype, during the publish step, the broker is provided with a copy of the service implementation, the requirements monitor and configuration files as shown in Procedure 3. This step is supervised (steps 1–3). Based on these three inputs, the set of requirements are translated to the list of fields and methods that would need to be monitored (step 4). Note that the service provider is never aware of the actual requirements and if the publish step is compromised no information about the intended requirements is leaked. During this step, reference measurements are also stored for the software stack upto the requirements monitor (steps 8–9). Finally, representative traces are stored by instrumenting the service implementation (steps 5–7).

As described in the next section, a client triggers a trust verification step by sending a request to certify to the trusted third party. On receiving such request, the trusted third party initiates a trust verification step by sending a nonce to the service provider. For every transaction, the broker generates a unique nonce to guard against replay attacks. This nonce is sent to the requesting provider with appropriate credentials to certify that it is from the correct third party and not an imposter. These steps are shown in detail in Procedure 4.

During the service, the trust analyzer monitors and creates a trace of the execution containing the field and method information identified during the initialization

Procedure 3 Publish step for brokers' end

Input: Provider Address: providerAddr, Service Id: Id
Input: Property as Criterion: sliceCriterion

1: impl = recvImpl(providerAddr, Id);
2: reqMonitor = recvMonitor(providerAddr, Id);
3: confFiles = recvConf(providerAddr, Id);
4: progPoints = slice(impl, sliceCriterion);
5: instImpl = instrument(impl, progPoints);
6: progTrace = genTrace(instImpl, reqMonitor, confFiles);
7: store(progTrace);
8: checksum = computeSHA1Hash();
9: store(checksum);
10: send(providerAddr, Id, progPoints);

Procedure 4 Bind step for brokers' end

Input: Provider address: providerAddr
Input: Service Id: Id
Input: Client address: clientAddr

1: Nonce n = generateNonce();
2: send(providerAddr, Id, n);
3: Trace t = receive(providerAddr, Id);
4: result = verifyPK(t,signature);
5: **if** result == false **then**
6: reportViolation(clientAddr, Id);
7: **else**
8: result = checkFresh(t.Nonce);
9: **if** result == false **then**
10: reportViolation(clientAddr, Id);
11: **else**
12: result = checkTrace(t);
13: **if** result == false **then**
14: reportViolation(clientAddr, Id);
15: **end if**
16: **end if**
17: **end if**

phase. Authentication server is responsible for verifying this trace. To that end, after receiving the signed data from the provider, the public verifying key of the AIK is used to verify the data and it's signature (step 3). At this stage, two violations can be detected. First, if the data from the provider and it's signature do not match, it represents corruption of provider's information and hence a violation of trust. Second, if the nonce contained in the data is not fresh, it represents a possible replay attack attempt by the provider. Client is notified of such violations (steps 6, 10).

On successfully verifying the data and the nonce, the next step is to verify the trace (step 12). The traces from two different executions may differ due to modifications of service implementation as well as difference in input values. Thus, exact equivalence may not be used to report

violations. Instead, we use the trace received from the provider to check if it contains a sequence that violates the requirements. In our example, persistence and serialization of data is assumed to be a violation of requirements, thus presence of an API call that persists or serializes data will be reported as violation. The main purpose of recording the trace during the publish step is to optimize this checking. The recorded trace represents the normal execution of the service implementation. An abstract form of stored trace is then used to quickly identify aberrations in current trace. Such behavior is then checked in detail.

Finally, if all checks pass, the system configuration is checked. If any of these checks fail, the clients are notified of a violation, otherwise an assurance is sent to the client.

D. Client's Implementation

For every transaction involving sensitive data, the client sends an attestation request to the broker and also simultaneously invokes the web service. The integrity of a transaction is assumed only if an assurance is received from the broker. On receiving this assurance the results of the service can be used as intended. These steps are shown in detail in Procedure 5. Note that the standard find and bind steps are omitted for simplicity.

Procedure 5 Bind step for client's end

```
1:  initTransaction();
2:  requestCert(brokerAddr,providerAddr,Id);
3:  sendRequest(providerAddr, params);
4:  result = receiveCert(brokerAddr,Id);
5:  if result == violationNotification OR timeOut then
6:      abortTransaction();
7:  else
8:      commitTransaction();
9:  end if
10: return;
```

E. Illustrative Example

We now use the example web service from Section I to illustrate the working of our prototype. Figure 3 shows this example. This payment processing service consumes the credit card number, the card validation code (cvc) and the purchase order, as the input from the client and produces confirmation number as the output. In this example, the client is unaware of the fact that the web service provider has processed the client's input for adversarial purposes and that it has stored the input 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.

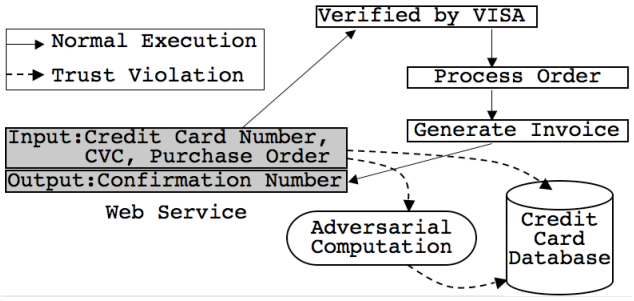

Fig. 3. An Illustration of Trust Violation

We created a sample implementation of this service. The web service is invoked from a web browser of another machine. The web service accepts the credit card 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 our prototype to observe the definition and use of data fields carrying sensitive data. The trace produced by the monitor is shown in Figure 4. 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
```

Fig. 4. Output trace for the original service

Since the monitor is also in the service provider's environment it can be compromised to not report the violations. We mimicked the compromise by instrumenting the monitor to ignore any violations that might have occurred. Such a requirement's monitor can be used by a service provider to report wellness when all is not well with the web service. This is when our technique can falsify a wrong claim of the service provider. We used our prototype to validate the *monitor* to ensure that it is not compromised or changed. Two cases of requirements violation are possible:

- The monitor detects a violation in the web service.
- The monitor itself is compromised and hence the violations are not reported. Such a compromise is detected by the hardware-based trust mechanism.

1) Detecting Requirements Violation: This class of compromise can be detected by current approaches for

requirements monitoring (e.g. [8], [9], [10], [11]). Using similar techniques, our prototype requirement's 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
(?,?,?)")
...
```

Fig. 5. Trace for compromised web service

Figure 4 shows the trace generated for the clean-room web service and the Figure 5 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 monitor checks the execution trace for the presence of calls to persistence APIs, it detected this violation.

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. One such case is presented in Table I in which one of the library files of the monitor was altered using bytecode instrumentation.

TABLE I. Comparison of TPM Measurements

File	160-bit SHA1 Hash of Genuine Monitor	160-bit SHA1 of the Modified Monitor
codemonitor.license	6476...DB8F	6476...DB8F
Connect.class	9F2B...A638	9F2B...A638
	...	
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 I presents the abbreviated list of files of monitor being monitored. 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* was altered without the knowledge of the broker.

F. Summary

In this section, we evaluated our claim that the realization of our technique is feasible using currently available hardware and software platforms. Our prototype although limited serves to satisfy our feasibility claims. It also points to interesting directions for further investigation such as design of a specification language for representing the requirements, more efficient mechanisms for monitoring such requirements on the service provider's end.

V. Evaluation: Potential Utility

The goal of this section is to validate the potential utility of our technique. To that end, two properties are important. First, that our approach should be effective in detecting compromises that the previous techniques couldn't detect. Second, that is should not have prohibitive overhead.

To analyze these properties, we evaluated our prototype using some standard subject web services. The subjects for our case study were selected from the web service implementations available from the Apache Axis distribution. Table II briefly describes the web services used for this case study and the sections of the service implementation that were traced by the requirements 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 V-B 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 such monitoring is negligible and thus, the proposed architecture efficiently monitors the integrity of the web services.

A. Violations

The class of compromise involving only the violation of requirements can be detected by current approaches for requirements monitoring (e.g. [10], [11]). Using similar techniques, our subject monitor was also able to give the execution trace for methods that caused either persistence or serialization of data. In case of such violations, the broker will notify a breach of trust to the client.

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

TABLE II. 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.

integrity of the web service is a function of the integrity of the requirements monitor. We presented one such case in Table I in Section IV-E, in which one of the library files of the monitor was altered. Since the monitor itself was being monitored, such violations were detected.

B. Overhead of Monitoring

Table III 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 II. These values are the averages of the time taken to execute the service over several client requests. The overhead due to CodeMonitor was 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. To work around the problem, we wrote custom AspectJ aspects [36] to monitor the same sections of the service implementation. As Table II shows this achieved a much better performance, which serves to show that if instrumentation techniques with less overhead are applied our approach is likely to have negligible overhead. This serves to validate our claim that web services can be monitored for integrity without a tangible time lag in responding to the client's request.

VI. Related Work

In this section we discuss closely related ideas. For reader's convenience these ideas are categorized along three key areas: techniques based on trust computing group's initiative, approaches that provide distributed attestation functionality for web-services, and those that allow modeling of non-functional aspects of SOA.

TABLE III. 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

A. TCG based integrity measurement

Sailer et al. proposed a TCG based Integrity Measurement Architecture for Linux [12]. 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 [13]. 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 into the system. The advantage of this architecture is that it could verify integrity of a system up to its application layer (web server).

This 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 [37]. 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 requirement's 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.

B. Distributed Attestation Models

WS-Attestation, an attestation architecture proposed by Yoshihama et al. [38], also 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 at the provider's site. Further, the authors have mapped this model to WS-Trust [17] 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. Our approach goes beyond validating the remote platforms. Using our proposed architecture, platform along with the web services themselves can be monitored for integrity violations and compromises.

Katsuno et al. proposed a new model of a distributed coalition, called Trusted Virtual Domain (TVD) [39] 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 component that wishs 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 verifies 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 [40]. 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 [41]. 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. In comparison, our architecture can be used to ensure the integrity of the aglet server, which would then provide a basis of integrity for aglets.

C. Verifying Non-functional Properties for Service-oriented Architectures

Some approaches have recently been proposed to verify contracts for web services, as seen in the works of Kuo *et al.* [7], Baresi *et al.* [4], Barbon *et al.* [5], Mahbub and Spanoudakis [6], etc. These ideas focus on verifying the behavioral contracts as defined by the externally visible interface of the web services, whereas our work provides a technique for verifying such requirements that require inspecting the web service implementation via a monitor.

The focus of Kuo *et al.*'s approach is on facilitating a more concise representation of the message exchange protocols as Boolean formula associated with each exchanged message, which in turn helps verify whether a given message exchange is legal. On the other end of the spectrum are approaches to validate the functional and non-functional requirements of a web service such as by Baresi *et al.* [4], Barbon *et al.* [5], Mahbub and Spanoudakis [6], etc, which use dynamic monitoring to ensure that a service-oriented architecture is satisfying its requirements. These techniques rely on monitoring the functional interface, often during service composition, to determine conformance of a web service to its requirement. Non-functional requirements that can be verified by a monitor bootstrapped using our approach are not addressed.

Wada et al. proposed a UML profile to graphically model non-functional aspects in SOA so that they are

incorporated in the development phase [42]. 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, however, it does not help with verification of service implementations for existing service-oriented architectures.

Last but not least, Canfora et al. have presented a detailed analysis of the fundamental issues and solutions related to various perspectives of testing a service-centric model [43]. Our work addresses some of the challenges motivated by this work, but more work remains to be done.

VII. Discussion: Relation to the State of Practice in Service-oriented Computing

It would be important to put our work in the context of existing research and practice in service-oriented computing. To that end, in this section we first consider how our proposed architecture can be realized as a straightforward extension of existing publish-find-bind-execute paradigm. We then consider how our approach integrates with existing standards in service-oriented computing area such as WS-Security and WS-Trust.

Our approach requires minimal modification to the roles of the service providers and service brokers to provide additional capabilities to the clients. The service providers will need to deploy requirements monitors, if they intend to provide the additional assurance to their clients. This is not such an impediment, as service providers already go to lengths to provide assurances to clients, e.g. it is common to buy and install SSL certificates for assuring authenticity. Furthermore, such monitors are going to be the same for every client. Also, it is likely that interaction protocols with such monitor can be standardized in a way similar to other standards in service-oriented computing arena. Service brokers will assume an additional role of trusted-third party as discussed in Section III.

The traditional notions of publish-find-bind-execute will only be slightly modified to account for our approach. During the publish step, in addition to the functional specification, service providers will send the capabilities of their deployed monitor and integrity measurements. During the find step, in addition to the desired functional properties a client will also send the desired non-functional policies to the broker as shown in Figure 2 in Section III.

The broker, which also acts as trusted-third party will use functional properties as well as desired policies to select a service provider. The available list of service provider could be first filtered by the functional specification. For each service provider in this filtered list, the broker would determine whether the capabilities of the deployed monitor satisfies trust and data integrity policies desired by the client. The broker will then send a list of such providers to the client as a response to the find request as shown in Figure 2 in Section III. Note that this simple model can be further enhanced by considering the trust history of service providers. A cost-benefits model could also be superimposed, where based on the price that the client is willing to pay for the service, a subset of service providers can be made available.

WS-Trust is used to secure the interactions with secure token exchanges and exchange of credentials. The value addition of our architecture to these protocol is in verifying the integrity requirements of the services as well. Just the security of message exchange between components involved in a transaction is not enough. Verification of integrity of involved services is also important along with making the transactions secure. Our proposed architecture can thus be used as an enabling platform for web service security standards such as WS-Trust and WS-Security.

VIII. Conclusion and Future Work

Verifying conformance to non-functional requirements is important for inspiring client's confidence in remotely-hosted web services. In this work, we proposed a technique, its implementation, and experimental validation, which serves to verify the integrity of a remotely-hosted requirements monitor. The implementation of the proposed technique was evaluated using the standard web services available with the Apache Axis Distribution. The evaluation demonstrated the feasibility of implementing our technique. It also demonstrated that our technique was effective in monitoring the non-functional requirements of the web services. Our current experimental results have looked at static checksum as a method of ensuring the integrity of the monitor. In the future, besides conducting an extensive evaluation of the overheads associated with this mechanism, we will also explore dynamic mechanisms.

Furthermore, we have only informally validated the claims for our approach. In future, we plan to formalize our approach that will allow us to provide more rigorous evaluation. This would include developing a core calculus for the TPM's machine model based on Spi calculus [44]. This semantics would account for the authentication, secrecy, and integrity properties of the TPM. Furthermore, a formal semantics for our approach can be built on top of this core calculus similar to the techniques proposed by Gordon and Pucella [45]. Some of this work is underway [34] and the reader is encouraged to visit the URL http://www.cs.iastate.edu/~tisa, where latest results from the Tisa project are regularly published.

Acknowledgements. This material is based upon work supported in part by the National Science Foundation under Grants CNS-05-40362, CNS-06-27354 and CNS-07-

09217. Thanks to Harish Narayanappa for help with some initial implementation. Authors would like to thank Robert Dyer, Youssef Hanna, Mehdi Bagherzadeh, and Rakesh Setty for comments on previous versions of this work. Thanks are also due to the anonymous referees from the IEEE Transactions on Services Computing editorial board for their careful review and suggestions on the draft.

References

[1] L.-J. Zhang and D. A. Grier, "Service oriented computing is overrated: Information infrastructure problems cannot be solved by service oriented computing (soc) alone. pro or con?" *Business Week*, November 2008.

[2] M. P. Papazoglou and D. Georgakopoulos, "Service-oriented computing: Introduction," *Commun. ACM*, vol. 46, no. 10, pp. 24–28, 2003.

[3] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web services description language (wsdl) 1.1," World Wide Web Consortium, Tech. Rep., March 2001.

[4] L. Baresi, C. Ghezzi, and S. Guinea, "Smart monitors for composed services," in *ICSOC '04*, pp. 193–202.

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

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

[7] 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*, pp. 915–918.

[8] M. S. Feather, S. Fickas, A. V. Lamsweerde, and C. Ponsard, "Reconciling system requirements and runtime behavior," in *IWSSD '98*, p. 50.

[9] S. Fickas and M. S. Feather, "Requirements monitoring in dynamic environments," in *RE '95*, p. 140.

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

[11] W. Robinson, "Monitoring software requirements using instrumented code," in *HICSS '02*, p. 276.2.

[12] 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*, August 2004, pp. 223–238.

[13] R. Sailer, L. van Doorn, and J. P. Ward, "The role of TPM in enterprise security," IBM Research, Tech. Rep., October 2004.

[14] 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*, Oct 2007, pp. 79–84.

[15] ——, "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.

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

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

[18] 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.

[19] "Trusted computing group," http://www.trustedcomputinggroup.org.

[20] "TPM main part 1 design principles specification version 1.2," www.trustedcomputinggroup.org/specs/TPM.

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

[22] S. Bajikar, "Trusted platform module (tpm) based security on notebook pcs - white paper," Mobile Platforms Group Intel Corporation, Tech. Rep., June 2002.

[23] 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*, 2003, pp. 3–10.

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

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

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

[27] A. Pnueli, "The temporal logic of programs," Weizmann Science Press, Jerusalem, Israel, Tech. Rep., 1997.

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

[29] "CodeMonitor^{TM}," http://www.tangentum.biz/.

[30] 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.

[31] M. Weiser, "Program slicing," in *5th international conference on Software engineering*, 1981, pp. 439–449.

[32] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," in *Symposium on Practical software development environments*, 1984, pp. 177–184.

[33] T. W. Reps and W. Yang, "The semantics of program slicing and program integration," in *TAPSOFT '89*, pp. 360–374.

[34] H. Rajan, J. Tao, S. M. Shaner, and G. T. Leavens, "Reconciling trust and modularity in web services," Department of Computer Science, Iowa State University, Tech. Rep. 08-07, July 2008.

[35] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 1999.

[36] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *ECOOP 2001*, pp. 327–353.

[37] 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*, 2004, pp. 79–84.

[38] 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.

[39] 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.

[40] 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)*, 2005, pp. 499–506.

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

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

[43] G. Canfora and M. D. Penta, "Testing services and service-centric systems: Challenges and opportunities," *IT Professional*, vol. 8, no. 2, pp. 10–17, 2006.

[44] M. Abadi and A. D. Gordon, "A calculus for cryptographic protocols: the spi calculus," in *CCS '97: Proceedings of the 4th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 1997, pp. 36–47.

[45] A. D. Gordon and R. Pucella, "Validating a web service security abstraction by typing," *Formal Aspects of Computing*, vol. 17, no. 3, pp. 277–318, October 2005.