Monitoring the Monitor: An Approach towards Trustworthiness in Service Oriented Architecture

Mahantesh Hosamani Harish Narayanappa Hridesh Rajan
Dept. of Computer Science, Iowa State University
{mahantesh, harish, hridesh}@cs.iastate.edu

ABSTRACT

The key notion in service-oriented architecture is decoupling clients and providers of a service based on an abstract service description, which is used by the service broker to point clients to a suitable service implementation. A client then send service requests directly to the service implementation. A problem with the current architecture is that it does not provide means for clients to specify, service brokers to verify, and service implementations to prove that certain desired non-functional properties are satisfied during service request processing. An example of such non-functional property is access and persistence restrictions on the data received as part of the service requests. In this work, we describe an extension of the service-oriented architecture that provides these facilities. We also discuss a preliminary implementation of this architecture and report preliminary results that demonstrate the potential practical value of the proposed architecture in real-world software applications.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Information flow controls; D.2.11 [**Software Architectures**]: Languages (e.g., description, interconnection, definition); D.2.5 [**Testing and Debugging**]: Monitors

General Terms

Design, Human Factors, Languages

Keywords

Service Oriented Architecture (SOA), web-service, verification, trust, client-side data privacy.

1. INTRODUCTION

Service-oriented architectures (or service-oriented computing paradigm) promote abstraction, loose coupling and interoperability of clients and services [6, 13, 14]. The key idea is introduce a published interface (often in the form of a description written in web services definition language (WSDL [5]) , which acts as a basis for communication between three types of entites : 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 [6, 13, 14]. 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 a GPS co-ordinate in a specified format and expect the service implementation to produce the address of the nearest hotel as a string to that GPS co-ordinate. The specification of input (GPS co-ordinate) and output (string containing the hotel's address) describe the functional requirements for this service.

Until recently specifying and verifying functional and non-functional requirements have not received much attention. Most recently Kuo *et al.* have discussed an approach for expressing and reasoning about functional requirements for service-oriented computing [9]. The focus of their 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. Another exciting result is described by Pathak *et al.*. They use a symbolic transition systems to help user specify a desired service goal in terms of functional and non-functional requirements and determine iteratively how this goal can be satisfied by selecting and composing a set of existing services [15]. The goal of these approach is to verify the construction of a service. On the other end of the spectrum are approaches to validate the functional and non-functional requirements of a running service-oriented architecture such as by Baresi *et al.* [3], Barbon *et al.* [2], Mahbub and Spanoudakis [11], which aim to ensure — using dynamic monitoring — that a service-oriented architecture is satisfying its requirements. The domain of this work is the later set of approaches.

In service-oriented architectures, services are often performed on machines that not owned and operated by users. Composition of services may happen on an entirely different machine all together. To monitor a service (or its composition) for functional requirements, such as "R1: the response given by the location-based hotel finding service shall be the closest hotel to the GPS co-ordinate specified by the client", it is sufficient to observe or test the interface of the service. On the other hand, to validate requirements such as "R2: the service shall not persist the GPS co-ordinate supplied by the client", it may not to be sufficient to validate just the external interface. This validation 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 [7, 8, 10, 17]). Nevertheless the key question remains, in a (possibly) untrusted domain who guarantee's the integrity of the monitor? In other words, who monitors the monitor?

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 ($M : SXI \rightarrow \{true, false\}$) running in a trusted environment, and a monitor that is similarly capable, but may be running in an untrusted environment ($M' : SXI \rightarrow \{true, false\}$), how can we validate that $M \equiv M'$ is always true.

What we do not do: we are not proposing an approach for runtime requirements monitoring, there are many other research papers on this topic e.g. [7, 8, 10, 17]. We do not propose to monitor functional requirements using our approach, they can very well be monitored by observing (or testing) the externally visible interface of the service.

To give the reader an idea of the problem with verifying a monitor in an untrusted environment without a root of trust, let us for a moment assume that a validation mechanism $V' : MXM' \rightarrow \{true, false\}$ exists. Now in order to answer this validation question, there must be a part of V' running in the same untrusted environment that can observe M' to compare it with M. If not, V' will depends 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.

We propose to use a hardware-based mechanism as a root of trust for such validation mechanism. Let us consider the example described in the previous paragraph. In this example, if we could be sure that there exist 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) [20, 19]. TPMs 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. In this work, we describe an architecture based on TPM to validate the integrity of a runtime requirement monitor, which will in turn facilitates trusted services.

Section 2 describes trusted platform modules, which form the basis of our proposed architecture. Section 3 describes our proposed architecture. The experimental evaluation of a prototype implementation conforming to this architecture is discussed in Section 4. Section 5 compares and contrasts our work with the related approaches. Section 6 discusses future work and concludes.

2. TRUSTED PLATFORM MODULE

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 [22]. 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. A TPM provides roots of trust for storage, measurement, and reporting of measurement.

Every TPM has a unique number assigned to it by the manufacturer called the *Endorsement Key*. 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. There is also a facility for on-chip public and private key pair generation using the inbuilt hardware Random Number Generator. This make it possible for the TPM to do encryption and decryption of data. The TPM also has a set of registers called *Platform Configuration Registers* 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 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 every reboot.

Over the past few years, 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) [22], 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) [12]. The hardware vendors are moving towards installing TPM on every computer that ships.

3. APPROACH

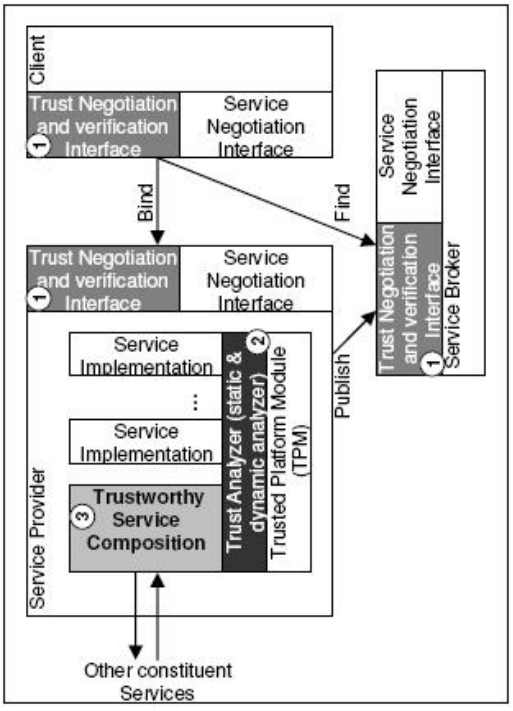

Figure 1: Our Proposed Architecture

Our proposed architecture is shown in Figure 1. The key additions to the standard SOA is a new interface that we call *trust negotiation and verification interface*. The purpose of this interface is to provide an abstraction for the clients to negotiate desired integrity levels and for brokers to verify that the service implementation is

indeed conforming to the desired service specification. The trust negotiation and verification interface between the service broker and the service provider 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.

We have implemented a very simple system based on this hypothesized architecture to show the feasibility of our approach. Our system is shown in Figure 2. To recapitulate briefly, in a SOA there are three main entities: the service provider, the service broker and the client (customer). In the context of this paper, the words client, customer and end user are used interchangably. The client contacts the service broker with a request and the broker directs the requester to the service provider. In our example system, the service broker also acts as the trusted third party. The monitor in this case is very simple, it verifies whether the service implementation on the service provider's side is genuine. This monitor can be directly implemented using the trusted platform module's primitives.

The trusted third party hosts an authentication server to authenticate whether the service implementation on the service provider's side is genuine. It does so by verifying whether the implementaion has changed since the last known deployment. For the purpose of this simple system, we are assuming that if the service provider had malicious intentions, the service implementation would be modified to either store or process the confidential customer data. It may not always be necessary; however, but we are deferring dynamic monitoring for future work. The goal of this architecture is to help the client to successfully complete the transaction with an assurance from the trusted third party that the service provider has not stored or processed the confidential data that were provided by the customer.

The algorithm for verifying the integrity is as described below.

1. A clean-room copy of the production software or the program is provided by the service provider to the trusted third party.

2. The authentication server on the trusted third party takes integrity mesurements by computing the 160-bit hash of important configuration files, source files and class files of the web-service implementation in a specific order using the inbuilt Sha1 hash engine of the TPM. The TPM computes the new 160-bit hash value by computing a SHA1 over the current 160-bit hash value. In this way any number of files can be measured into the same *Platform Configuration Register (PCR)*. These measurements are stored in the Authentication Server for future reference.

3. The authentication server sends an ordered list of files to the TPM on service provider's side. The TPM computes the 160-bit hash of these files in the given order and sends it across to the authentication server.

4. Each time the authentication server receives the 160-bit hash from the TPM, it compares this hash with the reference value stored at the time of clean-room measurement of the software.

5. Even if there is a slight difference in any of the measured files, there will be significant variations in the calculated SHA1 hash value from that file onwards. [25] claims that it takes 2^{69} units of time to find SHA1 collisions implying that collisions are very rare. Hence, there will be significant variations in the computed SHA1 hash values and these variations can be detected easily.

6. If the most recent hash value is different from the original reference hash value, the service broker which is also the trusted third party can warn the customer of this fact before the transaction itself.

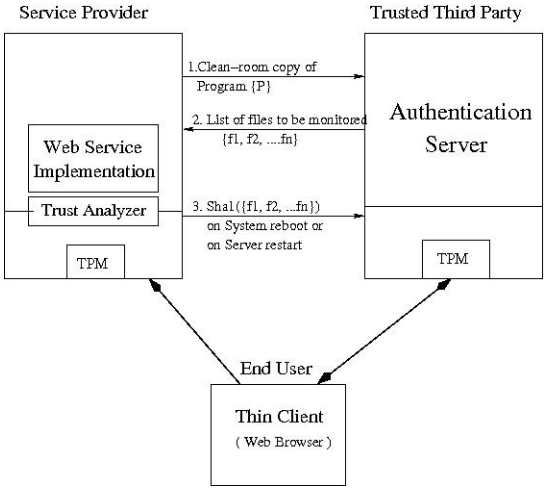

Figure 2: An Implementation of the Proposed Architecture

In the TPM, the PCRs are automatically reset to zero at the time of system reboot. The measurements made by the trust analyzer on the service provider's side are stored in a specific PCR of the TPM. The contents of this PCR is encrypted using the public part of the *Attestation Identity Key (AIK)* of the TPM of the trusted third party before sending data to it. 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. Thus, whatever data is received from the trust analyzer is trustworthy. The TPM of the trusted third party decrypts the PCR data using the private part of the AIK. This ensures that the hash value sent to the authentication server cannot be tampered by the service provider.

The above steps are repeated when one the following happens on the service provider's side:

- The operating system reboots.
- The web server or the SOAP server restarts.
- A patch is applied to the software.
- The source code of the software is changed, even slightly.

For every web service, the broker only needs to store a 160-bit hash value. So, the amount of extra disk space required to do the above operation is negligible. For small files the inbuilt hash engine in the TPM can be used to compute the hash value. Whereas, for large pieces of data, it is advantageous to use a hash engine outside the TPM, as the TPM hardware may be too slow in performance for such purposes. This is the main reason why the entire software is not be measured by the authentication server or by the trust analyzer. Only parts of the software that deal with the handling of confidential data and the files that deal with the critical system configuration are measured.

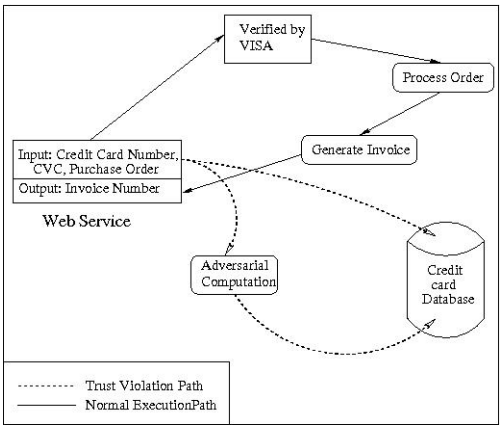

Figure 3: Example of Trust Violation by a Web Service

4. EVALUATION

This section describes an evaluation of our implementation for a simple web service dealing with financial transactions. This web service is described in the following subsection.

4.1 Example Web Service

Figure 3 depicts a common case of violation of trust by web services involved in financial transactions. The web service takes the credit card number, the card validation code (cvc) and the purchase order as 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 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.

4.2 Experimental Setup

The architecture was implemented using two Dell Precision 390 stations each having Intel Core2 Duo Processor @ 1.86 GHz and 2 GB of RAM. Both the stations have a TPM (Version 1.2) manufactured by Atmel Corporation, embedded in them. The Atmel TPMs in these stations have 24 PCRs each. One of the stations is assigned the role of a service provider while the other plays the role of a trusted third party. We used the *tpm4java* [23] for developing our trust analyzer to monitor and measure the implementation on the service provider. The Java library *tpm4java*, developed at TU-Darmstadt, Germany, 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 2-1.1.1 running on Windows XP Professional operating system.

4.3 Experiment

For evaluating this architecture, we created the web-service that carries out an online transaction for a customer as described in Section 4.1. The web service is invoked from a web browser in another machine. The customer gives the credit card number and the card validation code along with the list of items to be purchased. On the implementaion side, we have two files *Order.java* and *Process.java* that contain the code for carrying out this transaction and for giving back an invoice number as the output. There are two versions of these files. The first version is exactly according to the web service specification. Whereas, the second version is very similar to the first one except that it has been altered to store the credit card numbers locally.

The first column of the Table 1 lists the names of the files in the web-service implementation. The second column in the Table 1 shows the 160-bit hash values of PCR #10 during clean-room measurements of the software. The third column shows those measurements that were obtained after the source-code has been altered. It can easily be observed that the hash values in the third column starting from the entry corresponding to the file *Order.java* are all different 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 file including *Order.java* has been altered without the knowledge of the trusted third party. The list of files that were monitored, which includes log files, class files and executables, is long and only a subset of this list is published in this paper to demonstrate the viability of the concept. Thus, our architecture can detect a violation of trust in a web service implementation and can produce evidence for the same. A web service deployed in such a setting can claim to be "trust preserving".

5. RELATED WORK

Ever since the 1970s, efforts have been made to produce secure operating systems [21] 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 [1]. 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 also incorporates the capability to recover from integrity failures using replacement modules. Thus, it can guarantee that the system initializes to a secure state. Microsoft has incorporated a feature called Secure Startup in the Longhorn version of Windows [12]. Secure Startup has the capability to ensure that the PC running Longhorn starts in a known good-state. Another important contribution of this work was towards preventing denial of service attacks. 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.

In 2003, Grafinkel et al. proposed Terra, a virtual-machine based platform for Trusted Computing. Terra allowed multiple applications with diverse security requirements to run simultaneously on the same hardware. 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. Terra can give digital certificates for all of the software running on the virtual machines, to the third party for verification. However, it is not possible to selectively measure

Table 1: TPM Measurements for a Genuine and a Malicious Service

File	160-bit SHA1 Hash of Genuine Program	160-bit SHA1 of the Modified Program
../EchoHeaders.jws	34f6....32b7	34f6....32b7
../SOAPMonitorApplet.java	422b....b03a	422b....b03a
../SingleOrder.jws	fd53....77b5	fd53....77b5
../StockQuoteService.jws	b377....590b	b377....590b
../fingerprint.jsp	a841....09c2	a841....09c2
../happyaxis.jsp	2c7f....4030	2c7f....4030
../i18nLib.jsp	2a5f....e883	2a5f....e883
../index.html	524b....d1db	524b....d1db
../index.jsp	0302.....6ab9	0302....6ab9
../Order.java	***f354....14a3***	***3d12....fccb***
../Process.java	d679....2e8d	*9821....6490*
../charset.conv	8279....db20	*7fbf....f784*
../httpd-autoindex.conf	24bc....11af	*3da1....2c05*
../httpd-dav.conf	0600....79cb	*ea20....ddb1*
../httpd-default.conf	cb78....b514	*aeec....8bc7*
../httpd-info.conf	e86d....d676	*cb42....65c0*
../httpd-languages.conf	edd9....1fae	*a591....9b95*
../httpd-manual.conf	5526....40c7	*7701....9d58*
../httpd-mpm.conf	5d28....9ac3	*e062....5a41*
../httpd-multilang-errordoc.conf	6526....7a2e	*5f59....a0cf*
../httpd-ssl.conf	528b....579f	*b3e5....5215*
../httpd-userdir.conf	1337....4ba1	*e020....11e5*
../httpd-vhosts.conf	1be5....108a	*3101....229a*
../httpd.conf	c9fd....4aa7	*4fff....b99f*
../magic	f16c....e125	*ab63....6b9c*
../mime.types	7996....4748	*17a4....8059*

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 untrusted 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. Then, it is 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 [18]. 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 [24], a user-centric run-time information flow architecture. Information flow systems such as this allow untrusted 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.

In 2004, Sailer et al. proposed a TCG based Integrity Measurement Architecture for Linux [20]. 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 [19]. 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 cryptographic measurements are stored in the 160-bit Platform Configuration Registers (PCRs) of the TPM. 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 in-

volving 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.

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 [4]. 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. We support this idea of self-testing by using the trust analyzer and extend the concept to include trust as one of the issues in a service-centric system. Our 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. Only initiative that has to be taken for guaranteeing trust is to leverage the TPM hardware that is already installed in the system.

Another related approach is Aglet [16]. 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 untrusted 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.

6. CONCLUSION AND FUTURE WORK

Existing security models for web-services mostly consider the following four security and trust issues in the service infrastructures. First question is whether the requesting entity, i.e. client, is who they claim to be. Second question is the whether the client is authorized to use the service. Third question is whether the data, i.e. service request and reply messages, exchange between the client and the service provider is protected from unauthorized access and from tampering. 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. Our proposed architecture for ensuring the integrity of requirement monitors is a step in this direction. Our current experimental results have looked at static checksum as a method of ensuring the integrity of the monitor. In future, besides conducting an extensive evaluation of the overheads associated with this mechanism, we will also look into dynamic mechanisms.

7. REFERENCES

[1] 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.

[2] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, pages 63–71, Washington, DC, USA, 2006. IEEE Computer Society.

[3] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 193–202, New York, NY, USA, 2004. ACM Press.

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

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

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

[7] M. S. Feather, S. Fickas, A. V. Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *IWSSD '98: Proceedings of the 9th international workshop on Software specification and design*, page 50, Washington, DC, USA, 1998. IEEE Computer Society.

[8] S. Fickas and M. S. Feather. Requirements monitoring in dynamic environments. In *RE '95: Proceedings of the Second IEEE International Symposium on Requirements Engineering*, page 140, Washington, DC, USA, 1995. IEEE Computer Society.

[9] 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: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, pages 915–918, Washington, DC, USA, 2006. IEEE Computer Society.

[10] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Monitoring and control in scenario-based requirements analysis. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 382–391, 2005.

[11] K. Mahbub and G. Spanoudakis. Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, pages 257–265, Washington, DC, USA, 2005. IEEE Computer Society.

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

[13] M. P. Papazoglou. Service -oriented computing: Concepts, characteristics and directions. In *WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*, page 3, Washington, DC, USA, 2003. IEEE Computer Society.

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

[15] J. Pathak, S. Basu, and V. Honavar. Modeling web services by iterative reformulation of functional and non-functional requirements. pages 314–326, Washington, DC, USA, 2006. IEEE Computer Society.

[16] A. Rezgui, M. Ouzzani, A. Bouguettaya, and B. Medjahed. Preserving privacy in web services. In *WIDM '02: Proceedings of the 4th international workshop on Web information and data management*, pages 56–62, New York, NY, USA, 2002. ACM Press.

[17] W. Robinson. Monitoring software requirements using instrumented code. In *HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9*, page 276.2, Washington, DC, USA, 2002. IEEE Computer Society.

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

[19] R. Sailer, L. van Doorn, and J. P. Ward. The role of tpm in enterprise security. Technical Report RC23363 (W0410-029), IBM Research, Thomas J. Watson Reserch Center, Yorktown Heights, NY 10598., October 2004.

[20] 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.

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

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

[23] E. Tews and M. Hermanowski. Projektvorstellung tpm4java trusted computing fur java. http://tpm4java.datenzone.de/trac/attachment/wiki/MRMCD/mrmcd101b-tc.pdf.

[24] 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: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

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