End-to-End Policy Monitoring and Enforcement for Service-Oriented Architecture

Mehdi Azarmi
Department of Computer Science
Purdue University
West Lafayette, Indiana
mazarmi@purdue.edu

Bharat Bhargava
Department of Computer Science
Purdue University
West Lafayette, Indiana
bbshail@purdue.edu

***Abstract*—A service-oriented architecture (SOA)-based application is composed of a number of distributed and loosely-coupled services which are interconnected to accomplish a more complex functionality. The main security challenge in SOA is that we cannot trust the participating services in a service composition to behave as expected all the time. Moreover, the chain of all services involved in an end-to-end invocation may not be visible to the clients. As a result, any violation of the client's policies could remain undetected. To address these challenges in SOA, we propose the following contributions. First, we propose a new end-to-end security architecture for SOA based on a dynamic composite trust model. To maintain the dynamic trust, we designed a trusted-third party service called *trust manager* component, which collects and processes feedbacks from the actual execution of services. Second, we developed an end-to-end inter-service policy monitoring and enforcement framework (PME framework), which is able to dynamically intercept the interactions between services at runtime and react to the potentially malicious activities according to the client's policies. Third, we design an intra-service policy monitoring and enforcement framework based on taint analysis mechanism to monitor the flow of information within services and detect and prevent information disclosure attacks. These two frameworks together can provide an end-to-end visibility and security in SOA. Finally, we have extensively studied the correctness and performance of the proposed security frameworks based on a realistic SOA case study in a cloud environment. All experimental studies validate that the practicality and effectiveness of the presented solutions.**

***Keywords*-End-to-End Security; Service-Oriented Architecture; Security Policy**

I. Introduction

Service-Oriented Architecture (SOA) is an architectural paradigm in software engineering that promotes reusability, interoperability, and scalability of software systems through composition of loosely-coupled services. In recent years, SOA and microservices (which follow similar goals) have gained a significant attention by the exponential growth of cloud computing as an enabling technology[1].

SOA opens the door into new security challenges that are not present in the traditional client-server architecture. Even though a significant amount of effort has been done to study and improve the security of web services, still majority of the proposed solutions address the security challenges of single services or client-server models. The root of this challenge is in the arbitrary service invocation patterns (in form of Directed Acyclic Graphs) that happen in SOA. On the other hand, since participating services in SOA belong to multiple organizations, service consumers cannot monitor or control them directly. Therefore, some services down in the invocation chain may access a malicious service without being detected.

For example, consider a chain of services S_1, S_2, and S_3, that service S_1 from company A accesses S_2, which is hosted in company B. Then S_2 invokes S_3 from company C. In this scenario, S_1 (based on its negotiated policy) expects service S_2 to perform the requested functionality by itself or by using another trusted service (e.g., service S_4 from company D). However, for some reasons (potentially, with financial motives), S_2 decides to use S_3, which is not trusted by client. In such cases, client has no way to know that its sensitive information are exposed to company C (information leakage). Similarly, company B may be hacked and the compromised S_2 sends the sensitive information to an unknown server. In such scenarios, service consumers have no visibility or control over untrusted service calls. Even though commonly used security protocols (such as SSL/TLS) can help to provide a point-to-point security, however, the lack of dynamic trust and end-to-end security makes it necessary to design a new monitoring and enforcement framework, which is able to verify the service invocations at runtime and ensure that service consumers' policies are enforced.

Attack Model. We assume the service infrastructure providers (cloud providers or service providers) are not malicious. Furthermore, the attackers may have the following capabilities: (1) attackers may tamper with the service binaries or configurations, (2) attackers may have full access to the in-transit messages. The proposed PME component will be deployed outside of the web service containers at the JVM (Java Virtual Machine) layer. Therefore, if attackers obtain access rights at the level of application server configuration, they will not be able to tamper with or bypass the PME component. We further assume either the software

[1]Throughout this paper, the term *SOA* is used as an umbrella term, which includes microservices and other service-oriented patterns without focusing on any specific technologies (e.g., Enterprise Service Bus).

stack (OS and JVM) and the PME component are protected by hardware-based *trusted computing* techniques or any tampering with them can be detected by remote attestation techniques [1]. These assumptions are realistic and several successful attempts has been reported in the literature [2]. Finally, client-side weaknesses and infrastructure-specific (cloud and virtualization) security problems are outside of the scope of this paper.

Summary of Contributions. The key contributions of this paper are as follows:

1. A new policy-based security architecture for end-to-end security in SOA. We designed an extensible and flexible security architecture, which addresses the main SOA security challenges (information disclosure and lack of end-to-end visibility and accountability) in a holistic way. One of the main component of this architecture is a new trust management system that defines quantified and dynamic trust for SOA invocation patterns based on actual execution of services. The main design principles that we followed in designing the proposed security architecture are as follows: technology agnosticism (being extensible to multiple technology stacks, e.g., Java, .NET, and multiple protocols e.g., REST and SOAP) and real-world practicality (ability to work with legacy systems and closed source services; requiring no direct changes in those systems). These design principles enable the incremental deployment and backward compatibility, which are crucial factors for industrial adoption. We have leveraged aspect-oriented programming as a tool to meet these principles in designing our security framework.

2. End-to-end policy monitoring and enforcement mechanisms for SOA. To achieve this goal, we designed and implemented an inter-service and an intra-service policy monitoring and enforcement mechanisms. Even though our implementation target the Java-based web applications, however, since we use AOP paradigm, it practically can be extended to all major languages too[2]. These two mechanisms complement each other to provide an end-to-end protection. The inter-service mechanism tracks the flow of information *among* services. On the other hand, the intra-service mechanism, complements the operation of the first mechanism by extending the information flow control *within* services. This end-to-end framework enables the service consumers to define generic XACML policies and then detect or prevent the policy violations (such as illegal information disclosure). In addition to detection and prevention, the proposed mechanism is able to *correct* certain malicious activities right before happening.

3. Comprehensive experimental studies. Majority of the previous research works in SOA security either have mainly focused on the theoretical aspects of their ideas or they only have opted for simplified and unrealistic proof of concepts. However, in this paper, we have spent a considerable amount of time on designing practical approaches and experimenting with realistic scenarios.

The rest of this paper proceeds as follows. In Section II we discuss the proposed architecture for end-to-end policy monitoring and enforcement in SOA. Section III describes the design of an end-to-end inter-service policy monitoring and enforcement framework. Section IV, discusses the design of an intra-service monitoring and enforcement mechanism based on taint analysis concept. In section V, we evaluate and discuss the proposed solutions. Section VI covers the related work. We conclude in Section VII.

II. Proposed Security Architecture

As we discussed in the previous section, the main security challenges in SOA are the lack of end-to-end accountability and visibility. In this section we propose a new security architecture that addresses these challenges through dynamic trust management and end-to-end policy monitoring and enforcement mechanisms. The main components of the system (as depicted in Figure 1) are as follows:

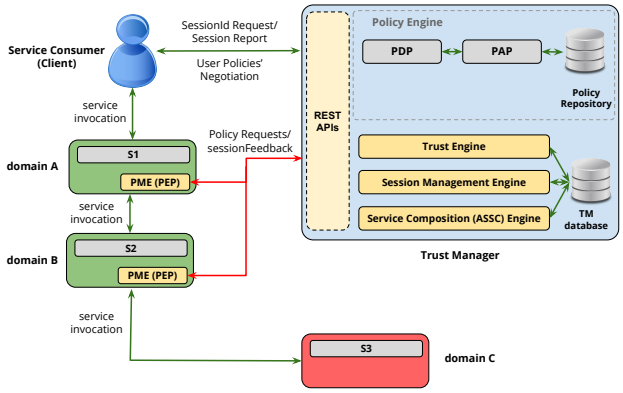

Figure 1. Proposed end-to-end architecture for SOA security

Policy Monitoring and Enforcement component (PME). PME components, that are deployed across the service providers, are responsible for monitoring the behavior of individual services at runtime in order to detect certain malicious behaviors (e.g., invocation of untrusted external services or leakage of private data). More formally, these components act as inlined reference monitors [3] and policy-enforcement points (PEPs) [4]. We have leveraged aspect-oriented programming (AOP) paradigm to make the PME components *non-bypassable* and *tamper-proof* against the potentially malicious target services. Each PME component monitors its target service for a set of predefined events (e.g., execution of method calls that are responsible for external service invocations) and then encapsulate the event and the

[2]https://en.wikipedia.org/wiki/Aspect-oriented_programming#Implementations

corresponding context (e.g., the target service address) and send them to the Trust Manager as a feedback message. In addition to passive monitoring (policy monitoring), PMEs are able to block the execution of services at critical points and ask the trust manager for permission (policy enforcement). The response from the trust manager instructs the PME component to either prevent the service execution or to permit its normal execution (or even correct the malicious behavior if possible). Further implementation details are presented in section III.

Trust Manager. Trust Manager is a trusted third-party (TTP) entity the plays a key role in the proposed architecture. In fact, this component is composed of a few other components: session management engine, trust engine, policy engine, and ASSC (secure service composition engine). To facilitate the integration of the proposed architecture into modern web-services, we have exposed all functionalities of the trust manager through REST APIs, which makes the trust manager easily accessible to the PME components and system administrators, and service consumers. We briefly explain the role of each sub-component in the following.

Session Management Engine. Session management component is responsible for creating and maintaining end-to-end sessions for the requests originated from the clients. These global sessions are created dynamically from individual session feedbacks reported by PME components.

Trust Engine. The main responsibility of the trust manager service is to collect security events (session feedbacks from PME components) and calculate and maintain the composite trust dynamically for a given set of services in SOA. In fact, the trust manger service maintains three categories of trust: individual service trust (which reflects inherent trustworthiness of a service based on its execution history), session trust (which reflects trustworthiness of a specific session based on current behavior of all participating services during this session), and composite trust (which represents the expected trustworthiness of a composite service starting from a certain service based on all collected execution histories and the interconnection structure of all accessible services). A client may query the expected composite trustworthiness of calling a service before the actual invocation or it can query the trustworthiness of a service session after making the request. The internals of the trust engine and the composite trust algorithms are discussed comprehensively in [5].

Policy Engine. This component is responsible for evaluating the policy requests (made by PME components) against the user-provided policies. We have leveraged WSO2 Identity Server as an open-source policy engine. This policy engine implements XACML 3.0 standard [4], which is a modern, open, and expressive policy language. Once the trust manager receives a request from a PME component, it extracts the context and creates a XACML request to be sent to the policy engine. The XAML policy response (permit, deny, etc.) will be used to select an enforcement strategy and create a response back to the PME component.

Secure Service Composition Engine. ASSC component provides a secure and adaptive service composition for SOA. It leverages an efficient algorithm to find the most trustworthy service composition among available services in multiple categories while meeting the cost and delay constraints of the overall composition. The details of this component is outside the scope of this paper and is available in [5].

Component Interactions Figure 1 illustrates the operation of the proposed framework for a reference scenario. In this scenario, Client can communicate with the trust manager to upload and enable its policies at any time. Later, for every request, client sends a request to trust manager asking for a session through a REST API (`/createsessionid/{clientId}`). Once a request succeeds, trust manager creates a session, which will be responsible for collecting the corresponding session feedbacks. SessionId and ClientId will be used by Client (as custom HTTP headers) to make the service request to S_1. Each service may belong to an independent service domain. If a service domain does not allow the deployment of a PME component (which is managed by a neural trusted-third party), it will be considered as a boundary of trust in SOA as the architecture will not be able to keep track of the requests. The PME component in service domain A intercepts the HTTP request and its context at the S_1's entry point and creates a new context, which later will be used to expand the end-to-end session (by sending a session feedback to TM). Right before invocation of S_2 by S_1, the PME component intercepts the target URL and sends a session feedback `(sessionId, invokerKey, invokedKey, metadata)` to TM. The `metadata` includes extra contextual information (for example, whether information is going to be leaked by this request). This session feedback will be used by trust engine and trust session management components in the TM. The session feedback call is asynchronous in the monitoring scenarios (to improve the performance). However, in enforcement scenarios, it will be synchronous and PME stops the execution of service and waits for the response from TM. This process will be repeated for the rest of the chain similar to previous step, unless it gets blocked by PME as a part of enforcement policy. If an invocation is not blocked by PME, the response of the SOA invocation is returned to the user. After the invocation is finished, client can send a request to the trust manager service to get the session report. Session report includes session trust, and potential policy violations.

III. Inter-Service Policy Monitoring and Enforcement

In this section, we discuss the internals of the inter-service policy monitoring and enforcement (PME) framework, which intercepts and controls the service invocations

among services.

A. Designing AOP-based Monitoring and Enforcement Mechanisms

As we discussed in section I, we use AOP as an implementation tool to meet our design principles. For designing the inter-service PME framework, we have followed four stages as follows:

1. Identifying triggers. In this step, we identify a list of potentially dangerous classes and their relevant methods that could be misused to perform malicious activities in the target service. In Java, the triggers could be selected from the APIs provided by standard JDK or third-party libraries. Potential triggers could be communication APIs (networking, RMI, JMS, etc.) or file access APIs. Identification of these trigger classes can be automated through traversing the package structures and class files in the service jar files by `javap` disassembler[3].

2. Designing corresponding pointcuts. Pointcuts are similar to regular expressions that match certain locations in the execution of a program.

3. Extracting contextual metadata. Context is any information that could help with deciding the maliciousness of the operation (e.g., the address of the called service). The contextual metadata is automatically extracted from arguments of the target methods through AOP mechanisms.

4. Taking proper actions. In enforcement scenarios, if a policy is violated, then the specified action by trust manager must be taken. The details of the enforcement strategies are discussed in the next subsection.

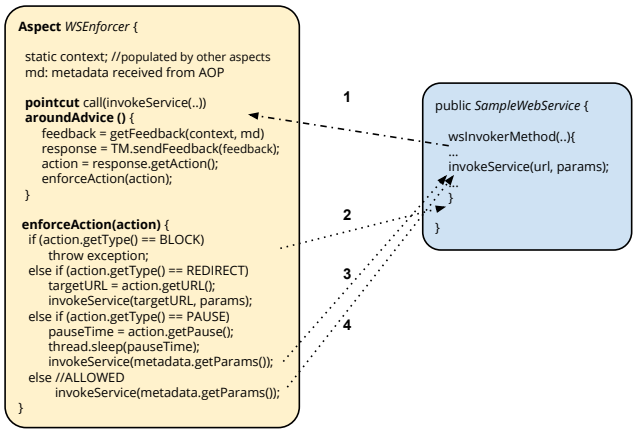

Figure 2. Enforcement of policies for service invocations using AOP

Figure 2 illustrates the internals of the inter-service PME framework. In this figure, `SampleWebService` uses `wsInvokerMethod()` to invoke another web service. To intercept this event, we design an aspect called `WSEnforcer`. Every aspect has a pointcut definition (here matches invocation of `invokeService()` method) and one or more advices (which specifies what should be done once pointcut is matched). We define an `around` advice which encapsulates the target service invocation. Right before the invocation of `invokeService()`, the AOP framework finds out that the `(call (invokeService(..)))` pointcut is matched and then replaces this method with `aroundAdvice()`. The `aroundAdvice()` extracts the context and sends a session feedback request to TM. Next, trust manager prepares a XACML request (which includes the maintained trust values) and calls the policy engine. Finally, trust manager processes the response of policy engine and responses back to the PME component. At this point, the `enforceAction()` is called to apply the enforcement strategy (shown by 2–4, depends on the enforcement strategy).

[3]https://docs.oracle.com/javase/8/docs/technotes/tools/unix/javap.html

B. Enforcement Strategies

In response to the policy violations, trust manager chooses an enforcement strategy (which could be originally defined by clients through *obligation*s in their XACML policies). The main *enforcement strategies* are as follows: aborting the service, redirecting to an equivalent trustworthy service, delaying the service (for throttling or rate control), rearranging the topology dynamically through ASSC, or executing the service normally.

IV. Intra-Service Policy Monitoring and Enforcement

Even though interception and control of invocations among services (as in inter-service PME framework) can significantly improve the security of SOA, however, it could be too coarse-grained in some scenarios. The challenge is that a service may invoke another service without disclosing any sensitive information. Similarly, the untrustworthy results produced by this invocation may not be used to produce the final response for the client. In none of these cases, it is justifiable to punish a service by reducing its trust value. On the other hand, the standard security techniques, such as firewalls and access control mechanisms, are not able to detect and prevent sophisticated information leakage attacks [6]. In fact, it is beyond the scope of such mechanisms to determine whether information is handled correctly *within* a service. Information leakage is the root cause of privacy violation, which has been an enormous concern in the recent years. In this section, we propose the intra-service PME framework, which addresses this challenge through fined-grained tracking of client's private information inside services. Combining both inter- and intra-service PME frameworks enable the end-to-end control on flow of information in SOA.

The intra-service PME framework is inspired by taint-analysis mechanism. However, taint analysis frameworks usually operate at the binary level (e.g., x86 or x64 instructions). We believe, the semantic gap between such frameworks and the web services is huge and these frameworks cannot be leveraged effectively. To address this challenge, we designed and implemented a taint analysis framework based on AOP. This framework enables working with high-level constructs instead of machine instructions. In addition to filling the semantic gap, this framework incurs a very low overhead compared to low-level taint analysis frameworks.[4]

The intra-service policy monitoring and enforcement is designed and implemented as an efficient taint analysis framework in Java using JBoss AOP. In the following, we define the taint source, taint sink, and taint propagation policies in the proposed framework.

Taint sources. Taint sources of a service are entry points, where sensitive information enters that service (e.g., reading a file or a network packet). In this paper, we define taint sources as methods that handle the incoming requests and process the private information such as credit card information received from clients or other services. Figure 3 shows the AOP pointcut that identifies the taint source in the ticket reservation scenario, which presented in section V. In this pointcut, `maliciousReserve(..)` is an entry point method to the service. Once e request arrives to the target service, the `processTRService()` advice from the `edu.purdue.cs.soa.pme. TaintAction` aspect class will be called to mark this object as tainted. Similarly, we can write pointcuts to define taint sources for files or servlet-based services (by intercepting `FileReader.read()` or `HttpServletRequest.getParameter()` methods).

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

Figure 3. Taint source pointcut in ticket reservation scenario.

Taint sinks. Taint sinks are exit points inside a service, where sensitive information might leave the service. For example, this information could be stored in a file or sent out to a network. We define the taint sink as any method that invokes another service. Figure 4 shows an AOP pointcut that identifies the taint sink in the ticket reservation scenario.

[4]Even though, the *taint* word could be used as a potentially dangerous input to a program, however, since the operation of our framework is similar to taint analysis, we use the same terminology for tracking the sensitive information.

This pointcut intercepts the calling of `post` method from jersey REST framework. Similar to the taint sources, we can easily define new taint sinks based on the scenario requirements.

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

Figure 4. Taint sink pointcuts for intra-service monitoring framework

Taint propagation policies. If an instruction f uses a tainted object x to produce object y, then y becomes tainted (or taint is propagated from object x to object y). Taint propagation depends on the semantic of the operation and has transitive property: $y = f(x) \wedge z = f(y) \rightarrow z = f(x)$.

We consider two categories of propagation policies. The first category addresses taint propagation through method calls and field accesses. The *aspect* corresponding for a method call, extracts the method arguments of the invoked method and uses the internal taint data structures to find whether any of them are tainted and then applies other taint propagation policies to track the taint flow inside the method body. If the return value is tainted, then the object that receives the output of this method will be tainted. The *aspect* which is responsible for field access, verifies whether the target field is tainted, and then taints the object that accesses this field accordingly. If the right-hand side of an assignment is not tainted and is assigned to a tainted object, we will untaint that that object by removing from the list of tainted objects.

The second category of taint propagation policies, tracks the information flow in the operations that access multiple fields. Specifically, we are interested in String fields and the operations applied to them as most of the sensitive private information (e.g., credit card numbers, SSNs, and addresses) are represented by `String` objects. In Java, String type is represented by four different classes: `String`, `StringBuilder`, `CharArray`, and `StringBuffer`. Therefore, we have defined pointcuts to process all of these classes. For each string operation, we call the corresponding advice from the `StringProcessor` aspect. To support real world web services, we defined propagation policies for all relevant constructors and methods.

V. Implementation and Evaluation

In this section, first, we describe a ticket reservation case study, which is used to conduct the experiments. Second,

we evaluate the performance of the inter-service and intra-service PME frameworks based on two different EC2 cloud testbeds. Finally, we briefly discuss the security evaluation of the proposed frameworks.

A. Ticket Reservation System Case Study (TRS)

To study the end-to-end security challenges in SOA, we have developed a tool in Java, which enables an arbitrary SOA application to be plugged-in or an arbitrary SOA scenario to be automatically generated. In the automatic mode, we can model a SOA application as an acyclic graph in the GUI, and then automatically convert it to an actual SOA testbed (based on RESTful JAX-RS web services) for studying attacks and trust algorithms. Further details are available in [5]. In this paper, we have developed a realistic SOA-based ticket reservation system (TRS) (a screenshot is shown in Figure 5). The TRS applicatoin is composed of three categories of services: ticket reservation services (denoted by TR), airline services (denoted by AA, DL, and UN), and banking services (denoted by BN). In this application, client invokes a ticket reservation service (TR_1 or TR_2) to search for a flight and receive a list of relevant airfares. To perform this operation, the TR service invokes three airline services to get the available airfares from each service. In this scenario, we assume there are three airlines and each airline is represented by three equivalent airline services provided by different companies[5] The TR service assembles all the received airfares from three airline services and send a response back to the client. In the next operation, user can choose an airfare from the list of returned airfares and initiate a confirmation operation. Once TR service received a confirmation request, it forwards it to a corresponding airline service and the target airline service communicates with a banking service (specified by BN) to finalize the payment. In this Figure, C_i represents a class of service (equivalent services with similar functionalities). The CoS is useful in redirection and dynamic service reconfiguration scenarios.

B. Performance Evaluation

We have conducted our experiments in two testbeds based on Amazon EC2 cloud. In both testbeds, we have used 64-bit Amazon Linux instances in the US-w2 (Oregon region) region and the client (Apache Benchmark tool) is located in a t2.micro instance (1 vCPU, 1GB memory, EBS). We have deployed application services, policy engine, and trust manager service in t2.small instances (1 vCPU, 2GB memory, and EBS storage) for the first testbed and in c3.xlarge instances (4 vCPU, 7.5GB memory, and SSD storage) for the second testbed.

We have used Apache Benchmark tool as a client to generate requests with varying levels of concurrency. In each run, we sent 1000 requests to the TR service. The number of concurrent requests sent to the ticket reservation service (TR) was varied from 1 to 8. This scenario measures the performance of the confirmation operation (chain of $client \rightarrow TR \rightarrow AL \rightarrow BN$).

[5]*AA* stands for an American Airline service (which provides relevant airfares from an American airline), *DL* stands for a Delta airline, and *UN* stands for a United airline service.

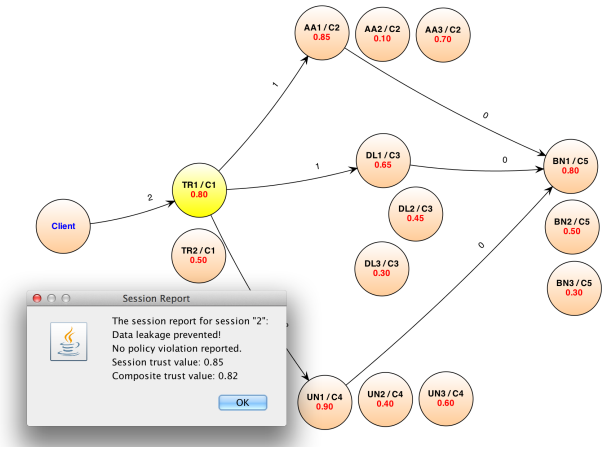

Figure 5. A screenshot of the ticket reservation system (TRS) that reports a data leakage attack prevention.

Response time. Figure 6 compares the average response times of the TRS case study in both of the cloud testbeds for the baseline (with no security measure), inter-service PME framework, and intra-service PME framework. The first observation is that in the first testbed the response time for each scenario increases almost linearly based on the concurrency factor. The reason is that in this testbed there is only 1 vCPU and all concurrent requests have to wait to get CPU time for processing. The second observation is that the inter-service and intra-service PME impose a noticeable overhead compared to the baseline scenario. The reason behind this overhead is that the PME component makes *blocking* calls to the trust manager service and waits until it receives the response (i.e, enforcement action). These calls triggers the invocation of policy engine by trust manager service, which adds to the response time. However, we have implemented the session feedback of the monitoring scenarios asynchronously (in separate threads), which decreases the overhead in multi-core systems. We can further observe the overhead of intra-service PME is not noticeably higher ($1.7 - 2.9\%$ higher) than the inter-service PME framework, which shows that the AOP-based taint analysis is very efficient. The results of the second testbed (shown with lighter colors) are significantly improved. For the concurrency-level 4, the response time for the taint analysis has improved around 67% compared to the first testbed. This speedup is very close to the ideal possible speedup (75% for 4 CPUs). The other observation is that since we only have access to 4 vCPUs, once we increase the concurrency level, the requests will be backlogged,

which increases the response time significantly. However, for concurrency level 1, the improvement is not significant. The reason is that the extra vCPUs are mainly useful under a parallel workload and they are underutilized in this scenario.

For the concurrency-level 4, the response time is around $60 - 68\%$ less than the first testbed. The improvement is slightly worse for the concurrency level 8, as the number of vCPUs are 4, and the requests will be backlogged.

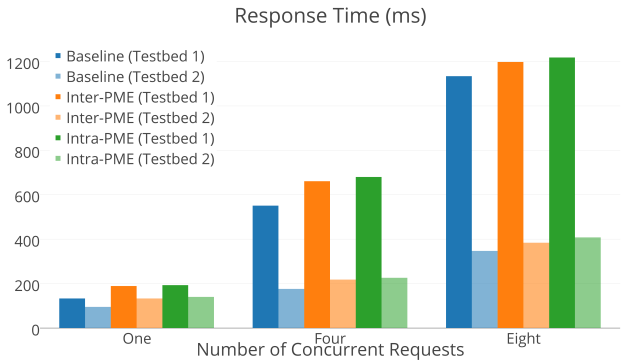

Figure 6. Response-time of three scenarios (baseline, inter-service PME framework, and intra-service PME framework) under multiple concurrency-levels on two Amazon EC2 testbeds for the ticket reservation system (TRS).

Throughput. Another performance metric that complements the response time is throughput, which represents the average number of served requests per seconds. Figure 7 shows that unlike the response time, the throughput of all scenarios in the first testbed are fairly stable and do not change significantly based on concurrency levels. The reason is that even though the backlog of requests in higher concurrency levels affects the response time, however, the utilization of a single vCPU remains similarly high. On the other hand, in the second testbed, the throughput of all three scenarios increases around 300%. These improvements confirm the scalability of the taint analysis framework as the number of CPU cores increases.

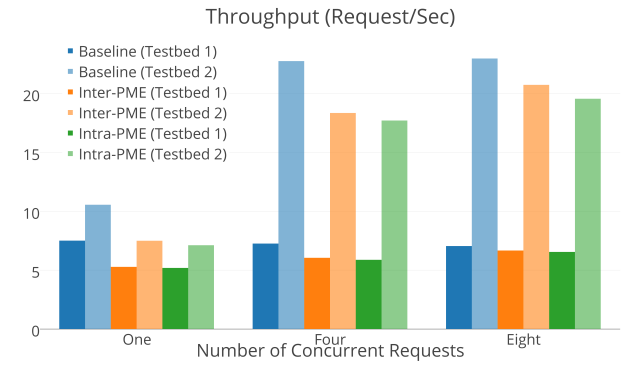

Figure 7. Throughput of three scenarios (baseline, inter-service PME framework, and intra-service PME framework) under multiple concurrency-levels on two Amazon EC2 testbeds for the ticket reservation system (TRS).

C. Security Evaluation

There are two ways for improving the security of a system, taking preventive actions and taking corrective actions. For the first approach, we make sure that the proposed framework is able to monitor the predefined policies and prevent any service invocation that violates those policies and discloses the private information. For the corrective approach, we enhance the trustworthiness of the SOA application through taking corrective actions (e.g., redirection or service recomposition based on ASSC).

We have tested the correctness of our framework based on several XACML policies with varying degree of complexities based on multiple custom attributes. These policies restrict access to services across a SOA application based on trustworthiness, whitelisting, blacklisting, and some application-specific parameters (e.g., price and date in the TRS scenario). These policies take advantage of *obligations* to define what action should be taken as a result of policy evaluation (e.g., permit, block, or redirect).

Using the UI, we can easily change the trust value of each service and see the impact of this change. For example, by dropping the trust level (which simulates a compromised service) of the UN_1 service from 0.90 to 0.27 (bellow the trust threshold), while the `redirect` policy is enabled, causes the automatic replacement of UN_1 with UN_3, which has the highest trust value. We also have designed a data leakage attack that once it is activated, it invokes a special REST API for a malicious reservation, which leaks the private data provided by client to an airline service. This experiment verified that the taint analysis framework is able to detect and prevent such attacks efficiently. Figure 5 is an screenshot of the session report, that shows a data leakage attack from TR_1 service is prevented. In this scenario, TR_1 decides to call the airline service while it supplies part of the client's private data. At this point, the taint analysis stops the calls and reports to the trust manager service that the TR_1 is going to leak the private information. When the trust manager service responds back with a *block* command, the taint analysis, terminates the current session.

VI. Related Work

She et al. describe an information flow control and access policy mechanism called SCIFC for SOA, which is based on delegations and pass-on certificates [7], [8]. The goal of this approach is to control the leakage of sensitive data while being transformed by intermediate nodes in a chain of services. The first shortcoming of this approach is its high overhead as it requires to validate certificates in every service along the service chain per request in both forward and backward directions. Another limitation of this approach is the lack of any practical enforcement mechanism. Moreover, the authors assume that the involved services are semi-trusted, which could be an unrealistic assumption as trustworthiness of services change dynamically and their status

may change from trusted to untrusted in a short time span. Finally, they only simulated the proposed approaches and they do not provide any realistic experimental experimental results. We have addressed these limitations by introducing two scalable PME frameworks that leverages a dynamic trust management system.

Aspect-oriented programming (AOP) was originally introduced by Kiczales et al. [9]. This paradigm extends the object-oriented programming by enabling the definition of independent crosscutting concerns. There are a number of web service QoS monitoring frameworks based on AOP (e.g., [10]) that are based on message interceptions and they use specific orchestration technologies (e.g., BPEL) and they do not address the security policies. Moreover, a large number of research activities (e.g., [11]) have been focused on techniques based on static program rewriting, which does not provide the flexibility of our proposed frameworks.

The AVANTSSAR platform[6], defines a few tools and languages to formally define and automate the validation of trust and security in SOA through model checking and reasoning techniques. However, these techniques are static and they require time-consuming modeling and precise system specification. Moreover, this approach requires access to the source code of services to be able to analyze them.

Dynamic taint analysis (e.g., DTA++ [12]), infer the information flow by observing the execution of a program. Most of the available taint analysis schemes incur too much overhead (e.g., 10–25 times overhead in [13]) to be used at runtime. We are interested in a solution, which is efficient enough to be utilized in the production scenarios in runtime. To address these limitations, we decided to design a new taint analysis framework for SOA using AOP. Leveraging AOP enables designing a framework which is transparent to the services and users.

VII. Conclusion and Future Work

In this paper we showed that using of composite trust models backed by service execution monitoring, provides an effective framework for end-to-end visibility, accountability, policy monitoring, and policy enforcement in SOA. Such frameworks enable SOA consumers to detect and prevent data disclosure attacks in a chain of services and adapt the SOA application to increase the overall security. The experiments presented in this paper, confirm that both inter-service and intra-service PME frameworks are practical, scalable, and effective.

Addressing the scalability of the trust manager service is not in the scope of this paper. However, we present a few ideas as a future direction. Even though the results of the second testbed shows that a multi-core system can improve the scalability of the proposed architecture, however, we can further improve it by distributing the trust manager service into multiple servers and partition the maintained states among them. In this solution, the high availability and fault tolerance can be achieved through Paxos or Raft consensus protocols. Furthermore, caching mechanisms (both server-side and client-side) can further improve the performance.

[6]http://www.avantssar.eu/

References

[1] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, 2011.

[2] R. Perez, R. Sailer, L. van Doorn *et al.*, "vTPM: Virtualizing the trusted platform module," in *Proceedings of 15th USENIX Security Symposium*, 2006, pp. 305–320.

[3] U. Erlingsson, "The inlined reference monitor approach to security policy enforcement," Cornell University, Tech. Rep., 2003.

[4] E. Rissanen, "Extensible access control markup language (XACML) version 3.0," OASIS http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-os-en.html, Tech. Rep., 2012.

[5] M. Azarmi and B. Bhargava, "An end-to-end dynamic trust framework for service-oriented architecture," preprint. Available at http://mazarmi.org/doc/trust.pdf.

[6] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[7] W. She, I.-L. Yen, B. Thuraisingham, and E. Bertino, "Security-aware service composition with fine-grained information flow control," *IEEE Transactions on Services Computing*, vol. 6, no. 3, pp. 330–343, 2013.

[8] W. She, I.-L. Yen, and et al., "The SCIFC model for information flow control in web service composition," in *IEEE International Conference on Web Services, (ICWS'09)*, 2009, pp. 1–8.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *ECOOP Object-Oriented Programming*. Springer, 2001, pp. 327–354.

[10] L. Baresi, S. Guinea, O. Nano, and G. Spanoudakis, "Comprehensive monitoring of BPEL processes," *IEEE Internet Computing*, vol. 14, no. 3, pp. 50–57, 2010.

[11] K. W. Hamlen, M. M. Jones, and M. Sridhar, "Aspect-oriented runtime monitor certification," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2012, pp. 126–140.

[12] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic taint analysis with targeted control-flow propagation." in *Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[13] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," *Internet Society*, 2005.