An End-to-End Security Auditing Approach for Service Oriented Architectures

Mehdi Azarmi*, Bharat Bhargava*, Pelin Angin*,
Rohit Ranchal*, Norman Ahmed*†, Xiangyu Zhang *, Asher Sinclair†, Mark Linderman†, Lotfi Ben Othmane‡
*Department of Computer Science
Purdue University
West Lafayette, IN, USA
{mazarmi,bb,pangin,rranchal}@cs.purdue.edu

†Air Force Research Laboratory
Rome, NY, USA
{norman.ahmed, asher.sinclair, mark.linderman}@rl.af.mil

‡Eindhoven University of Technology
Department of Mathematics and Computer Science
Eindhoven, Netherlands
l.ben.othmane@tue.nl

***Abstract*—Service-Oriented Architecture (SOA) is becoming a major paradigm for distributed application development in the recent explosion of Internet services and cloud computing. However, SOA introduces new security challenges not present in the single-hop client-server architectures due to the involvement of multiple service providers in a service request. The interactions of independent service domains in SOA could violate service policies or SLAs. In addition, users in SOA systems have no control on what happens in the chain of service invocations. Although the establishment of trust across all involved partners is required as a prerequisite to ensure secure interactions, still a new end-to-end security auditing mechanism is needed to verify the actual service invocations and its conformance to the expected service orchestration. In this paper, we provide an efficient solution for end-to-end security auditing in SOA. The proposed security architecture introduces two new components called *taint analysis* and *trust broker* in addition to taking advantages of WS-Security and WS-Trust standards. The interaction of these components maintains session auditing and dynamic trust among services. This solution is transparent to the services, which allows auditing of legacy services without modification. Moreover, we have implemented a prototype of the proposed approach and verified its effectiveness in a LAN setting and the Amazon EC2 cloud computing infrastructure.**

***Keywords*-Service Oriented Architecture; security auditing; taint analysis; trust;**

I. Introduction

Service Oriented Architecture (SOA) is a new design paradigm in software engineering which is characterized by *loose-coupling* among software components, called services. SOA allows rapid design of new applications by composing smaller special-purpose and heterogeneous services. In addition, SOA can serve as the unifying layer to integrate heterogeneous service components in both enterprise and military environments. *Web service* is a proven industrial technology that can be used to implement SOA applications. Web services make applications accessible over the standard Internet protocols. SOA paradigm allows any component to be wrapped behind a standard interface called *service bus* to provide a unified service interface to the service consumers. Service descriptions (*WSDL* files) are published by service providers and registered in the service registry, $UDDI$. The registered services could be discovered by potential users. Service discovery is based on matching the registered *WSDL* files with the required service specifications, provided by the user. Communication among services is performed using multiple protocols. The dominant protocol is the Simple Object Access Protocol (SOAP) that has associated web services standards and is mainly transported over HTTP [1].

Security is a challenging issue in service oriented architectures due to lack of end to end authentication and authorization. It is not possible to stop unwanted interception of messages by attackers. The unsecured SOA presents hackers with the ability to eavesdrop on SOAP message traffic and see information that may be private. On the other hand it is not possible to secure the unknown third parties in a SOA, because of the architecture's open nature. So, it is possible for secondary or tertiary services—the partners of your partners—to access an unsecured SOA.

To identify attacks, in most cases, it is not enough to rely on the existing standards. Even when attacks are detected, they are noticeable only within an interaction of two services or only inside a service. In some cases, the client may receive only a message that the service invocation did not succeed, without knowing the location of the failure. But in case of an illegal service invocation, the client will not even be notified.

More specifically, current SOA security solutions and Web service security standards have the following limitations:

- Web service standards are focused on transactions between only two communicating service end-points. They do not consider service composition. Information sent from a service provider to a service requester could leak due to information flow between services participating in providing the service, where the service is a composition of a set of services.
- They are based on the assumption that a service s_i that trusts a service s_j with its sensitive data trusts any service

that service s_j trusts. This assumption is not valid for applications that use sensitive data.
- External services are not verified or validated dynamically (uninformed selection of services by user).
- User has no control on external service invocation within an orchestration or through a service in another service domain. It means s_i does not have any control on the use of this data once it is received by service s_j . Service s_j can transform the data and disseminate it on its will.
- Violations and malicious activities in a trusted service domain remain undetected.

Contributions: This paper presents a new end-to-end security auditing architecture based on the introduction of two new key components into SOA: *runtime taint analysis* and *trust broker*. To summarize, our paper makes the following contributions:

- Designing a transparent service invocation control mechanism for SOA using service-level dynamic taint analysis.
- Designing a trust mechanism that tracks chain of service invocations. We designed the trust broker (*TB*) component that maintains information about trustworthiness of services and categorizes them. *TB* is used for dynamic validation and verification of services and keeps track of history of service invocations.
- Designing a secure end-to-end message origin authentication for Web service client requests and Web service providers to ensure confidentiality and integrity even in the presence of man-in-the-middle attacks. This solution is based on the common access card (CAC).
- Detecting compromised services and insider attacks by monitoring any illegal service invocation from services inside a trusted domain.

The remainder of this paper is organized as follows: in section II, we outline the proposed security architecture. In section III, we describe our prototype implementation. Moreover, a security and performance evaluation of the prototype is provided. We review the related work in section V. Section IV discusses future work and section VI concludes the paper.

II. Proposed System Architecture

This section presents the design and implementation details of various components of the proposed security architecture prototype.

A. Reference SOA Scenario

The end-to-end SOA infrastructure consists of a client making a request to the initial trusted services/domain and that service can make a service call to another service from a trusted domain or an untrusted public domain.

The definition of a trusted service, in the scope of this paper, is any of these items:

- A service consumer or client itself
- A Web service which is deployed in an environment controlled by an organization such as Air Force and a taint analysis module deployed in that service domain.

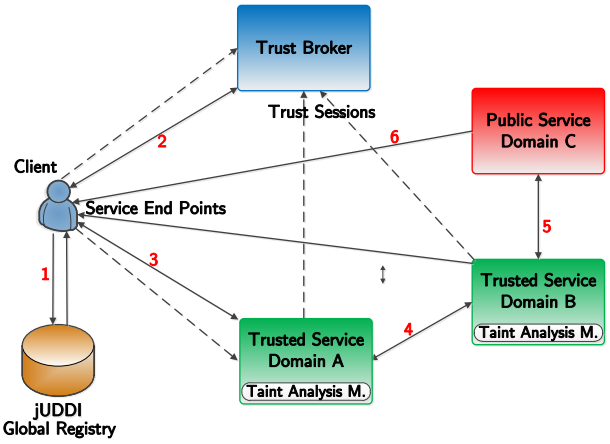

Fig. 1. SOA High-Level End-to-End Architecture and Reference Scenario

- Any other service domains that give permission for deployment of a taint analysis (*TA*) module.

All other services are untrusted services, which are those mainly under the control of external entities (Yahoo, Amazon, Google, etc.). Standard communication protocols used for these services are SOAP over HTTPS with WS-* support.

Figure 1 depicts the information flow in a reference scenario used in this paper. The information flow depicted in figure 1 is as follows:

1) The client queries the global service registry, UDDI with search parameters like service category and then UDDI returns a set of services matching the query to the client.
2) The client makes a selection from the set of services returned by the UDDI according to its requirements by comparing various SLA (Service Level Agreement) attributes; queries the *TB* (described in section II-B) with the selected set of services and gets back an ordered list from the *TB* categorizing services into various levels of trust (Certified, Trusted, or Untrusted). The client can then decide to contact a service that he chooses based on the returned trust levels.
3) After deciding on a service in the trusted domain A, the client registers the selected service (creates a session) in the *TB* to keep track of its session for end-to-end service invocation.
4) The service in the trusted domain A invokes another service in trusted domain B. During this invocation, the Taint Analysis (*TA*) module (described in section II-C) intercepts the communications and reports any illegal external invocations to the *TB*.
5) Step 4 is repeated: The service in domain B invokes a service in a public (possibly untrusted) domain C; this invocation is detected by *TA* and reported to the *TB*.
6) The response for the service request is sent to the client.

B. Trust Broker Subsystem

The *Trust Broker (TB)* is a trusted third party responsible for maintaining end-to-end security in a chain of service

invocations upon the request of a client and mediating security critical interactions between clients and services. The major functions of the *TB* are the following:

1) **Maintaining a list of certified services:** The *TB* maintains a list of services, which guarantee existence of a *TA* Module in their *enterprise service bus (ESB)* and compliance with a specific set of WS-* protocols. This is the highest level of trust that can be achieved by any service, as certified services allow for tracking of service invocations and ensure secure messaging. A service is listed as certified by the *TB* upon certification by an external trusted authority.
2) **Evaluating the trust level of a given service:** For services that are not in the certified list, the *TB* evaluates the trust level using a formula integrating various parameters including the history of interactions with that service, support for various standards (such as WS-*) and trust levels of the services in its service composition. This function of *TB* allows clients and other services (with dynamic service composition) to learn about the reputation of services before invoking them.
3) **Maintaining an end-to-end session of service invocations:** Upon a client's request, *TB* starts a session for the invocation of a service by that client, where the different services invoked from the start to the end of that session are logged by *TB*. For a particular session, invocations to other services (outsourcing of the client's request) are reported to the *TB* by the *TA* modules of the involved services. *TB* then evaluates the appropriateness of these service calls based on the properties of the invoked services (whether the services are certified, trust levels of the services, etc) and warning logs are created for that session if necessary. The logs of a specific session can be obtained from *TB* at any time before the log is removed from its database (removal of a session can be performed by the starting client or based on a timeout).

1) Trust Broker Structure: The Trust Broker was implemented as a Web Service in the Java 6.0 (`javax.ws`) platform. *TB* stores all data regarding sessions and services in a MySQL database, setup on the same machine as the service. The *TB* Web service offers the following public methods:

- `getTrustLevel(serviceKey)`: Given the key of a service (as registered in *UDDI*), returns the trust level for that service as calculated by the trust evaluation module.
- `createSession(trustClass, invokedService)`: Starts a new session for a client's invocation of a service identified with invokedService, where all services invoked in the end-to-end chain should have trust levels equal to or above that of trustClass. This method returns a unique session identifier, which needs to be passed along from the client to the invoked service and from one service to the other in the whole chain of invocations (Our solution for passing this identifier along is including it as a special header in each SOAP message in the invocation chain).

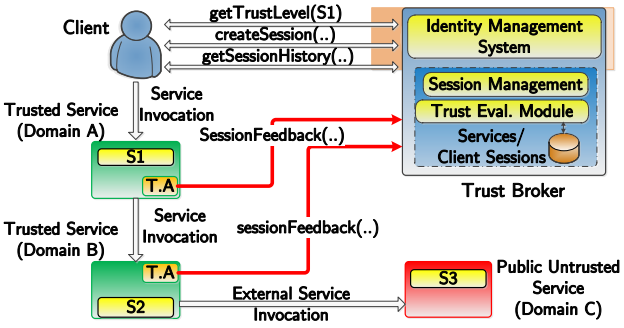

Fig. 2. Interaction of Taint Analysis with Trust Broker

- `getSessionHistory(sessionID)`: Returns the log of warnings for service invocations (if any) for the session identified with sessionID.
- `removeSession(sessionID)`: Removes the session identified with sessionID from the *TB* database.
- `sessionFeedback`: It connects the *TA* module to *TB* (presented in section II-C).

The structure of *TB* and its interactions with clients is shown in figure 2. Integration with *TA* module is discussed in section C.

2) Trust Evaluation Module: The *trust evaluation module (TEM)* of the *TB* calculates the trust level of a given service based on the history of previous service runs, the feedback from *TA* modules and WS-* support specified in Service Level Agreement (*SLA*). TEM queries the *UDDI* for calculating the trust value of a service, to obtain protocol compliance (WS-*) information regarding that service and the trust history of the service is retrieved from the *TB* database. TEM uses a *exponential weighted moving average (EWMA)*, where the recent feedbacks for a service are weighted more heavily than feedback further in the past. The trust value T is for a service S, with *SLA* trust value L, getting feedback F at time t is updated using the equation:

$$T_s(t) = \beta \times [\alpha \times T_s(t-1) + (1-\alpha) \times F] + (1-\beta) \times L \quad (1)$$

where α <0.5.

In the above equation, the constant β is the weight for the properties of the service such as its compliance with WS-* standards and α determines the significance of the past reputation of the service in comparison to a recent feedback received for that service.

The feedback parameter F in the equation takes on a negative value in the interval $[-1, 0)$ when the service in question misbehaves (e.g. invokes an external service with lower than desired trust level) and a positive value in the interval $(0, 1]$ when the service behaves as promised. This equation can be extended to include other parameters that could potentially affect the trustworthiness of a service such as the location of the service, authentication level of the service, response time of the service and composition of the service. The output of the equation is a real value in the interval $[0, 1]$.

C. Taint Analysis Subsystem

The taint analysis module monitors the activity of services (at runtime) and inspects the data exchanges (information flow) between them to detect certain events.

Runtime Service Monitoring: One of the design requirements of the *TA* component is to be transparent to the users. Therefore, users are not required to change their programs or deployment which is an important factor for adoption of a security solution. To achieve this goal, we need program instrumentation. In particular, extra instructions are automatically added to service implementation without the users' awareness. The execution of the instrumentation tracks the information flow in the execution. Intuitively, one can consider such instrumentation as hooks to the execution so that the *TA* component can gain control when certain events occur, which can be done at compiler time or runtime. The incurred overhead varies within the range barely noticeable to a few times slower, depending on the set of execution events monitored. In this project, we leverage *aspect oriented programming (AOP)* to achieve a special information flow control for auditing. The process is transparent and has small overhead.

Aspect Oriented Programming: Aspect Oriented Programming [2] frameworks instrument an underlying base program, but in AOP this purpose is more generic, to weave in any crosscutting functionality that should be factored out of the base program and not be replicated in the many locations in the program source where it is needed. A basic AOP model defines some specific fundamental *pointcut designators (PCD)*, which are features in the program execution where the advice of an aspect can be weaved in. A composition language allows a pointcut expression to combine and constrain these to define a pointcut, which is a set of program *joinpoints* (execution occurrences of the program features) that satisfy the expression, and where the advice will be executed. In existing AOP frameworks (JBoss AOP [3], AspectJ [4], and Spring AOP [5]), the fundamental pointcut designators are chosen somewhat pragmatically: they must be actually useful to an aspect programmer, but they must also be relatively practical to implement in the AOP system. Thus, in existing AOP systems, pointcut designators are typically points in the program where inserting instrumentation is *not too hard*; for example, method calls are very often used as one of the fundamental *pointcut designators*. JBoss AOP [3] and AspectJ [4] are powerful frameworks that implement AOP for Java programs. Its pointcut designators include method calls, method executions, and object field accesses.

Taint Analysis Implementation: There are many schemes of *TA* implementations in the literature that are heavy weight static and dynamic binary execution profiling based on non-web services environments. Tainting in this work is used to monitor information flow between trusted and untrusted services and report back to the *TB*. We selected JBoss AOP as a main framework for *TA* after investigating several technologies. Using JBoss AOP, we can monitor almost all classes and methods in the JBoss AS/ESB servers. This technology works very efficiently by using granular *pointcuts*. We instrument communication methods (used in service invocation) inside an action pipeline. This mechanism is illustrated in Figure 3. As shown in this diagram, all external service invocations are monitored and reported to the *TB*. For example, when an untrusted public service such as Amazon is going to be used as part of an Air Force service orchestration, the *TA* module monitors the activity and the data exchanges between these services. Monitoring is done mainly for two activities; one is to check the compliance of those domains (or services) to their registered *SLA agreement* as advertised in the public UDDI registry and the second is the consumption of their data into the trusted services domain. Reporting to the *TB* is done through a web service invocation to the *TB* server. The *TA* module invokes the `sessionFeedback(sessionID, invokerService, invokedService)` method, supplying the session identifier, the service key of the invoked service and invoking device.

The *TB* uses these reported incidents for:

- Monitoring information flow policies and tracking violations in a real-time SOA environment and logging for later reporting to clients.
- Decreasing the trust level of a service if it passes a message to a suspicious service.
- Adjusting trust levels for use in secure service composition.

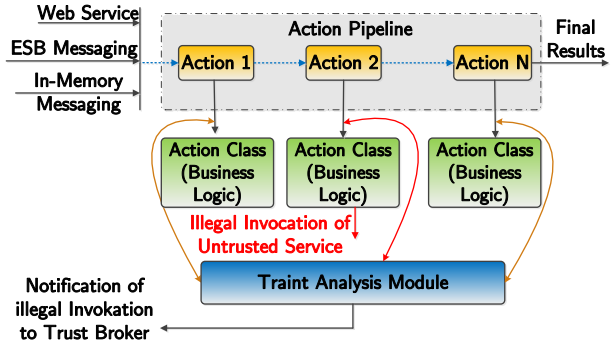

Fig. 3. Interaction of TA module with SOA action pipeline

Figure 2 illustrates the interaction of taint analysis with Trust Broker.

D. WS-* Security Standards

Many standards and protocols have been developed for secure message exchange among Web services, such as WS-Security, WS-Policy, WS-SecurityPolicy, WS-Trust and WS-SecureConversation that are often referred to as WS-* standards specification [6]. We take advantage of two WS standards in our prototype for achieving end-to-end security in the system: WS-Security and WS-Trust.

1) WS-Security: The end-to-end security architecture prototype uses WS-Security for end-to-end integrity and confidentiality of SOAP messages. WS-Security is a flexible

extension to SOAP to apply security to Web Services [7], published by OASIS. The protocol specifies how integrity and confidentiality can be enforced on messages and allows the communication of various security token formats, such as SAML, Kerberos, and X.509. Its main focus is the use of XML Signature and XML Encryption to provide end-to-end security.

The system prototype uses the Apache CXF Framework [8] leveraging WSS4J to provide WS-Security functionalities. WSS4J security is triggered through interceptors that are added to the services and the client. These interceptors allow performing the WS-Security related processes including: passing authentication tokens between services; encrypting messages or parts of messages; and signing messages.

2) WS-Trust: WS-Trust [9] is a Web service specification and OASIS standard that provides extensions to WS-Security. It defines a framework for requesting and issuing security tokens. Particularly, WS-Trust defines the concept of a *security token service (STS)*, a service that can issue, cancel, renew and validate security tokens, and specifies the format of security token request and response messages as well as ways to establish, assess the presence of, and broker trust relationships between participants in a secure message exchange.

In the system prototype, we used the open source implementation of STS known as PicketLink by the JBoss community [10]. The PicketLink STS does not issue tokens of a specific type. Instead, it defines generic interfaces that allow multiple token providers to be plugged in. As a result, it can be configured to deal with various types of token, as long as a token provider exists for each token type. It is deployed as a regular service in the JBoss ESB.

III. Experimental Evaluation

A. Use Case Scenario and Implementation Details

An emergency response use case scenario was implemented to demonstrate the end-to-end security auditing architecture in action. In this scenario, a chemical spill near an air base is announced and there is a need to evacuate its workers safely. A service consumer/client will need three different services to gather the information necessary to announce the evacuation plan. These services will include but are not limited to: 1) a trusted local service that provides shelter locations in the city, 2) a public weather service for determining the chemical plume direction, and 3) a public timer web service that estimates the time required for workers to be evacuated safely, which can possibly depend on another service. This scenario is highly generic, and the involved services can be re-arranged in any order to demonstrate an end-to-end secure service communication.

We use the term *client* for the end-user who issues the initial request. To fulfill the client request, the client domain issues a request to a service and gets the response back or that service may depend on other service(s) for parts of its response. The proposed solution at first, makes the initial service discovery (from client to the UDDI registry) secure. Second, it makes the communication between the client and the selected service secure, and finally, makes the service-to-service(s) communication secure (this approach recursively solves the end-to-end security problem by utilizing a combination of the *TA* module's and *TB*'s capabilities).

The action pipeline consists of three major components:

1) JMS Message Sender.
2) Weather Report Web Service
3) Evacuation Timer Web Service

JMS Message Sender: This performs the role of a Web service client by invoking the next component in the pipeline (Evacuation Timer Web service). The ESB that encompasses this provides the InitialContext from which the JMS Message sender can look up the queue where a message needs to be sent. The properties of the InitialContext provider are configurable. The queue name retrieved by the JMS message sender is deployed as part of the deployment of the ESB itself. This queue should also include a JMS-Gateway listener.

Once the JMS Message sender retrieves the queue name from the initial context provider, it creates a queue connection for the same. The parameter `zipcode`, that the Web service expects, is constructed as an `ObjectMessage` and sent to the queue connection which was created earlier. This invokes the next component in the pipeline which is a SOAP Message constructor, the purpose of which is to embed the webparam `zipcode` received from the JMS Message creator as a SOAP message that the Web services can understand. The SOAP message is then sent to the next major component.

Weather Report Web Service: This is an independent service which is deployed as part of its own ESB. This service can be deployed as an external service in a remote machine. Only the endpoints of the service are required for a client to communicate with this service. This Web service includes a Python script that uses Google Weather API to fetch the weather information for a given zip code. The response is sent back to the client invoking the Web method. The `jboss-esb.xml` service configuration file includes a SOAP client action that invokes the Weather Web service. The result is received by the next action in the `jboss-esb.xml` which gets the Web service response in the SOAP reply from the Weather Web service. Moreover, this component constructs another SOAP message that needs to be sent to the Evacuation Timer Web service. All these different components (SOAP Client action, Weather Web service, Response receiver) are all part of the same queue that ends with the Evacuation Timer Web service.

Evacuation Timer Web Service: The evacuation timer Web service takes as input the output from the Weather report Web service. This service is invoked using a SOAP action. This is a locally deployed service that requires a remote service (weather report Web service) for it to complete its operation. The deployment is again through an ESB container. Finally the SOAP message reply from the Evacuation timer service is returned to the client.

B. Performance Measurement in LAN

The runtime performance of the implemented prototype was evaluated in terms of the overhead caused by the security

components used in the system. The experiment setup in a local area network at Purdue University involved three machines (DAYTON, BOSTON and ROME), each with a JBoss ESB server deployment. The scenario involving the invocation of an evacuation timer described above was used for the services setup.

Hereafter, the term *baseline* is used to describe the experiments where no *TA* is involved in the service invocations and all data communication between different services as well as between clients and services is unencrypted (no WS-* standards are used). Figure 4 below sketches the setup for the baseline experiments. In this set of experiments, the ROME machine was used to host the Weather service, DAYTON hosted an Evacuation Timer service with no *TA* module installed in the ESB server and the service requests came from a client setup on BOSTON.

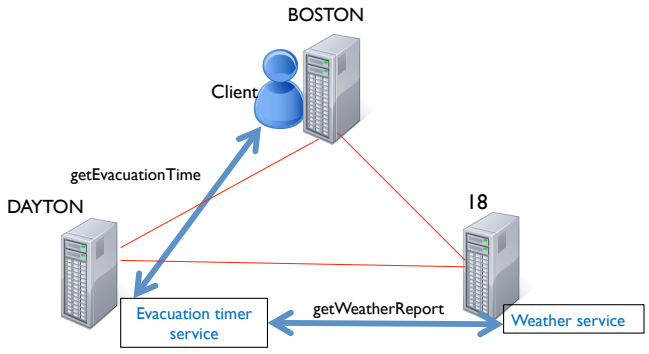

Fig. 4. Baseline experiment setup in the local area network

Figure 5 below sketches the setup for the *TA* experiments. In this set of experiments, the ROME machine was still used to host the Weather service, and DAYTON hosted an Evacuation Timer service with a *TA* module installed in the ESB server. In addition to hosting the client, BOSTON also hosted the *TB* service, which used a MYSQL database on the same machine.

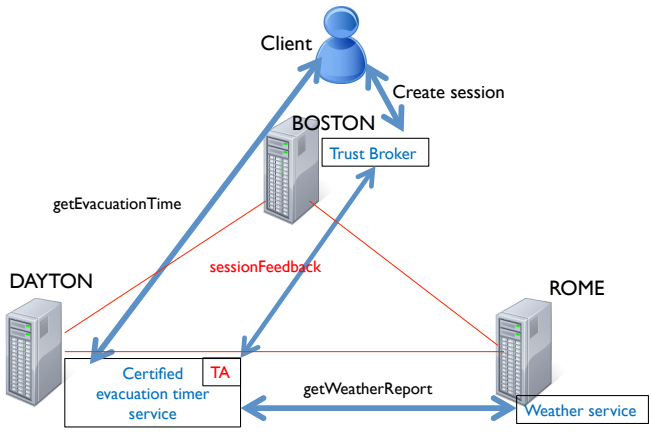

Fig. 5. Baseline experiment setup in the local area network

Experiments were conducted to measure the average response time for the client requests to the evacuation timer service under varying conditions. Figures 6, 7 and 8 below show the average response times for the invocations of the baseline and *TA* equipped evacuation timer services. In these experiments, the number of simultaneous requests sent to the Evacuation Timer services was varied from 1 to 16, which allowed for response times less than 1 second and 100% throughput (i.e. all requests were satisfied). Requests were sent to the service at the rate of 10 requests per second, i.e. the delay between the consecutive requests of a single client thread was set to 100 milliseconds and increased proportional to the number of simultaneous client threads. The experiments were repeated 10 times, each after a fresh start of the DAYTON server and the average for the 10 runs is reported in the graphs.

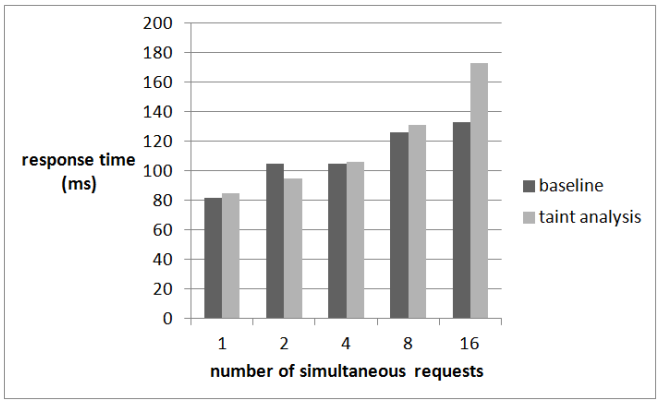

Fig. 6. Response time vs number of simultaneous requests for the first 300 invocations of the Evacuation Timer service

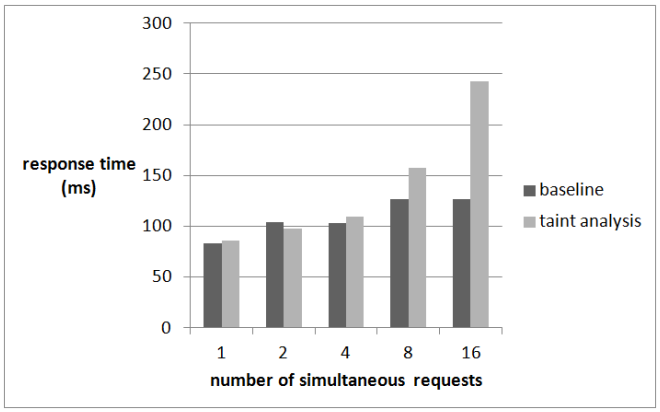

Fig. 7. Response time vs number of simultaneous requests for the first 350 invocations of the Evacuation Timer service

As we see in Figure 6, *TA* has negligible response time overhead up to periodic bursts of 8 requests and a small overhead for bursts of 16 requests. This is due to the fact that the *TA* module is loosely coupled with the evacuation timer service, requiring no change in the service code, hence presenting minimal overhead.

While for the first 300 requests the overhead of using *TA* is minimal, we see a more pronounced difference for 16 threads

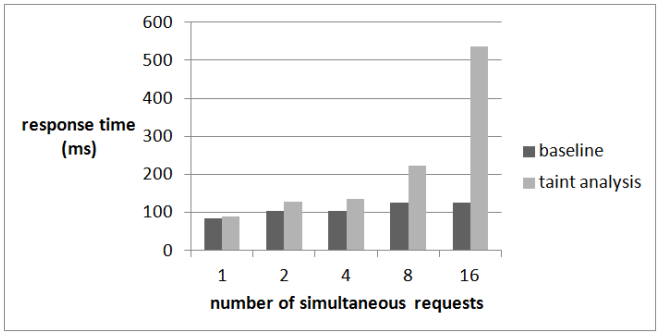

Fig. 8. Response time vs number of simultaneous requests for the first 400 invocations of the Evacuation Timer service

when the first 350 requests are considered. The overhead increases further for the first 400 requests. The most probable reason for this increase is memory problems due to the high strain on the server and load balancing should be considered in the case of expectations for high volumes of requests.

Figure 9 below shows the overhead caused by using WS-Security encryption with *TA* as opposed to using unencrypted data with *TA*. In these experiments, all data communication between the client and the Evacuation Timer service, between the Evacuation Timer service and the Weather service, and between the *TB* service and the *TA* module is encrypted with symmetric keys in compliance with WS-Security. The results suggest a difference of around 50 milliseconds in the response time, which is negligible overhead in most cases.

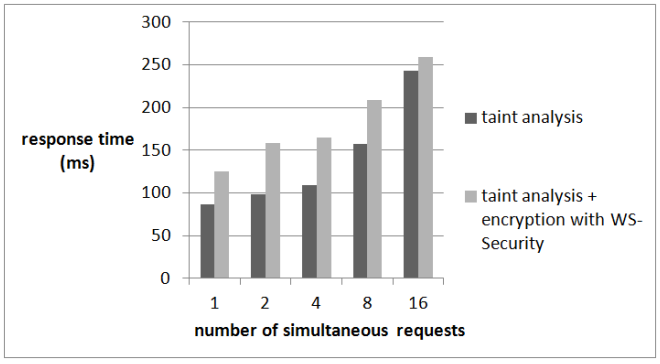

Fig. 9. WS-Security overhead on response time for invocations of the taint analysis equipped Evacuation Timer service

C. Performance Measurement in Amazon EC2

The Amazon Elastic Compute Cloud (EC2) (http://aws.amazon.com/) was used to study the impact of migration of the proposed end-to-end security solution to the Cloud. The experiment setups for the baseline and *TA* cases are shown in figures 10 and 11 below. In order to ensure that the services were deployed on different physical machines, large machine instances were launched in different availability zones of Amazon EC2 in the East region (Virginia) as seen in the figures.

Figure 12 below reports the average response times for the first 400 requests to the Evacuation Timer service for the

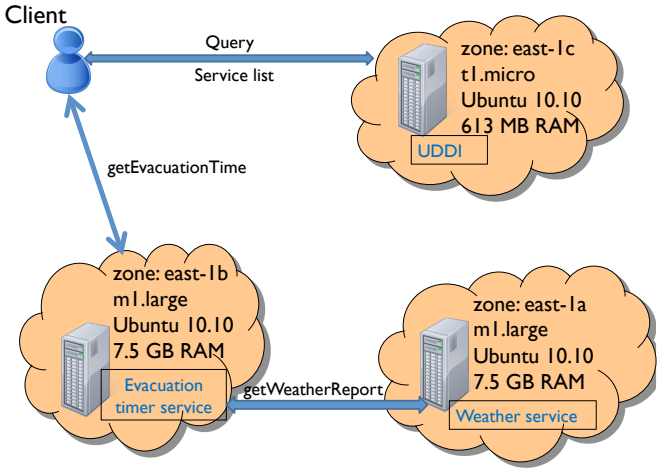

Fig. 10. Baseline experiment setup in the Amazon EC2

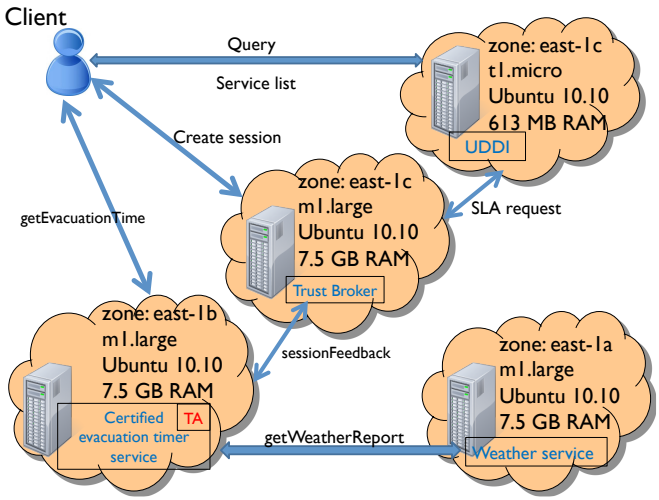

Fig. 11. Taint analysis experiment setup in the Amazon EC2

baseline and *TA* cases. As seen in the graph, the response times are still very close for up to 4 simultaneous requests. The overhead is somewhat larger, but still acceptable for 8 and 16 simultaneous requests. Experiments conducted in the Cloud were observed to have more varying results due to more uncontrolled conditions such as availability of bandwidth and other resources as the same physical machine can be shared by multiple virtual machines.

We also conducted experiments to measure the performance of the *TB* under increasing loads for the session feedback and session history methods. A large machine instance in the Amazon EC2 East region (east-1a availability zone) was used to host the *TB* for these experiments and all data communication between the *TB* and the client sending the requests was encrypted using symmetric keys in compliance with the WS-Security standard. Figures 13 and 14 below show the average response times (of the first 400 requests) for varying numbers of simultaneous session feedback requests to the *TB*. In the first set of experiments (Figure 13), the rate of requests was

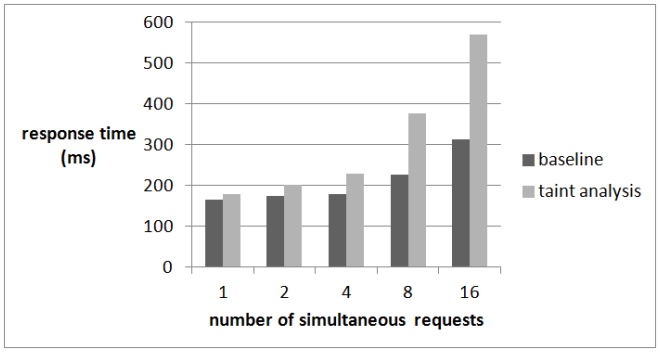

Fig. 12. Response time vs number of simultaneous requests for the first 400 service invocations in the Amazon EC2

kept fixed by setting the delay between consecutive requests by a single thread to 100 milliseconds and increasing the delay proportional to the number of simultaneous threads. The results for these experiments show that the *TB* is able to handle 64 simultaneous requests in around 150 milliseconds and 128 requests in around 200 milliseconds and the increase in response time for increasing number of threads generally follows a logarithmic trend despite the overhead from WS-Security based encryption.

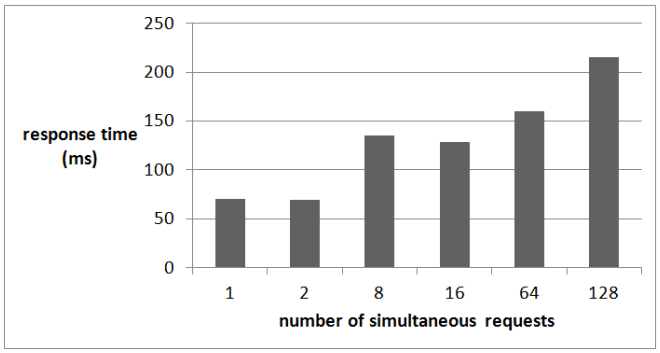

Fig. 13. Response time for fixed rate session feedback requests to a Trust Broker deployed in the Amazon EC2

In the second set of experiments (Figure 14), bursts of requests were sent at increasing rates, i.e. the delay between the consecutive requests of all client threads was set to 100 milliseconds. The results of these experiments show that the increase in the rate of requests causes a small overhead in the response time up to 64 client threads; however there is a big jump in the overhead after 128 client threads, at which point load balancing should be considered.

Figures 15 and 16 below show the average response times for varying numbers of simultaneous session history requests to the *TB*. In the first set of experiments (Figure 15), the rate of requests was kept fixed by setting the delay between consecutive requests by a single thread to 100 milliseconds and increasing the delay proportional to the number of simultaneous threads. The *TB* achieves 100% throughput up to 64 threads with average response times of less than 250 milliseconds.

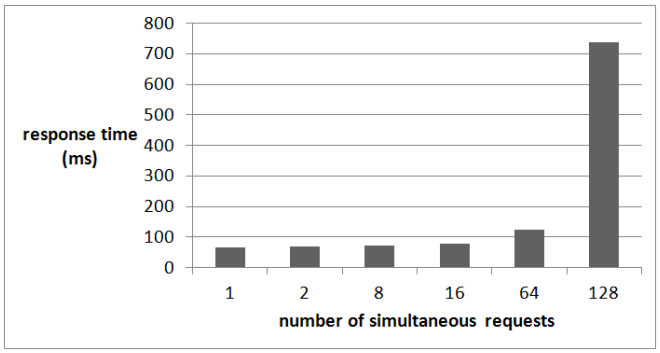

Fig. 14. Response time for the first 400 session feedback requests to a Trust Broker deployed in the Amazon EC2

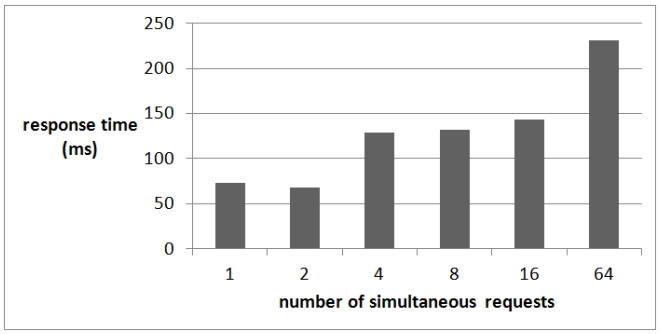

Fig. 15. Response time for the first 400 session history requests to a *TB* deployed in the Amazon EC2

In the second set of experiments (Figure 16), bursts of requests were sent at increasing rates, i.e. the delay between the consecutive requests of all client threads was set to 100 milliseconds. In these experiments the *TB* achieved 100% throughput up to 32 threads with average response times of less than 300 milliseconds. The response rate was observed to drop below 100% for 64 or more client threads, which requires load balancing.

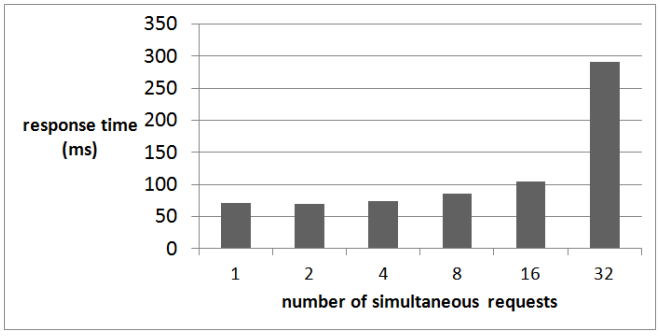

Fig. 16. Response time for the first 300 session history requests to a Trust Broker deployed in the Amazon EC2

D. Security Evaluation

Similar to many other networking applications, man-in-the-middle (*MITM*) attack [11] is a common attack in SOA. The XML Rewriting attack [12], [13] is a type of *MITM*

attack. In this attack, attackers manipulate the SOAP headers to replace or replay the SOAP messages without being detected in general web service domains by exploiting certain weaknesses of the XML Digital Signature and WS-Security protocol [13]. Our proposed solutions proved to detect these type of attacks successfully.

IV. Discussion and Future Work

Security Policy Enforcement: The goal of this paper has been to design a *security auditing* architecture. Therefore it takes a retroactive approach for external service calls and only reports the external invocation events back to the *TB*. However, it can easily be converted into a proactive mechanism to enforce client's (or originating domain's) policy. This could be realized by adding a policy engine (XACML [14]) to *TB* and employing *TA* module proactively. In such an architecture, *TA* module which acts as a *PEP* (policy enforcement point) always gets the relevant policies from *TB* which is considered a *PDP* (policy decision point). These policies are being applied at the corresponding domain, and if any of those policies is violated, then *TA* module would terminate the current service orchestration. Another variation could be sending the upcoming service invocation to the *TB* and requiring services in an invocation chain to get a confirmation for the next service they will invoke.

Cloud Computing Extensions: Experiments performed in the local area network as well as the Amazon Cloud (EC2) suggest that the proposed solution causes negligible overhead in terms of the service response time up to a certain load on the server, at which point load distribution should be considered. The same argument holds for the *TB* service as well; i.e. to avoid being a single point of failure prone to denial of service attacks, the *TB* should distribute its load over multiple servers. This makes the Cloud the best option for hosting the *TB* service. With elastic load balancing achieved by on-the-fly allocation of resources and creation/tear-down of virtual machines, a *TB* service in the Cloud will be able to meet the demands for different service request loads and prevent waste of resources in the case of decreased service traffic. Deploying services in the Cloud (in the case of using unreserved physical machines) brings up the question of multi-tenancy, which is a potential security threat. The proposed architecture partly mitigates the threats posed by multi-tenancy too, as in the case of a certified service being under attack, the *TA* module deployed on the server will report malicious behavior to the *TB*. Even in the case of a *TA* module under attack, it will be possible to detect that attack with a slight modification in the architecture. For that, the *TB* would need to wait on feedback from the *TA* module of every certified service, which is known to be invoked by a previous service and update the session history for that service call with a warning after a timeout period during which no feedback is received.

Investigating new threats for SOA-based systems in cloud computing environments: As discussed above, deploying services in the Cloud brings up the question of potential security threats due to multi-tenancy. In future work, we will investigate the possible effects of multi-tenancy on the proper functioning of the proposed solution. Experiments will be performed with multi-tenant Cloud servers, where attacks will be targeted from one virtual machine to another to disrupt the functioning of the TA component and solutions to the problems (if they arise) will be investigated.

V. Related Work

Security of service oriented architectures has been studied by many researchers. Authors in [15] and [16] address the security issues in SOA by focusing on web service standards. The approaches are based on the assumption that SOA services and their interactions are assured by mandating services to comply with standards based on stated security controls from NIST and DoD. The authors identify the complexity of certifying SOA services due to the difficulty in representation of security controls in web services standards specifications in a consistent manner for verification. The presence of a lot of standards, their cross-referencing, dependencies and inconsistencies between the various XML Schema specifications complicates the task further. In [16], they describe an approach to map security controls to specific XML elements of WS-*. In [15], they present a framework to model the security elements and define the proper SOAP message structure and content that each service must have, based on a security meta-language (SML) that models the security relevant portions of the standards for their consistent, comprehensive, and correct application. The problem is that compliance with standards doesn't completely secure SOA. Further, such approaches cannot provide protection against zero day attacks.

In [17] and [18] the identification of trusted services and dynamic trust assessment in SOA are studied. Malik [17] introduces a framework called *RATEWeb* for trust-based service selection and composition based on peer feedback. It is based on a set of decentralized techniques for evaluating reputation-based trust with ratings from peers. However they do not take into account initial service invocations and the secondary services in compositions. Spanoudakis *et. al.* [18] presents an approach to keep track of trusted services to address the compliance of promises expressed within their service level agreements (SLAs). The trust assessment is based on information collected by monitoring web services in different operational contexts and subjective assessments of trust provided by different clients (consumers) situated in specific operational context. They further address the issue of unfair ratings by combining user rating and quality of service monitoring. But they don't take into consideration user preferences and multiple quality of service parameters. They also don't show a final trust value. Approaches like [17] and [18] are not suitable for SOAs with a lot of services because the monitoring system would need to collect intensive information from a lot of peers and consumers, which would make it very expensive.

Generally, taint analysis has been a low level mechanism which has been used for binary program analysis [19]. But, on the other hand, low level taint analysis mechanisms lead to

a considerable overhead which is not suitable for real world services. Moreover, they are dependent on specific hardware architectures which is not suitable for real world deployment.

DIFC (Decentralized Information Flow Control) has been an active area of research in the past few years. Researchers in [20] [21] [22] have proposed different labeling mechanisms to secure applications from untrusted codes. Their approach needs a complete redesign of the OS which is not practical in the federated SOA settings. To overcome this problem, authors in [23] propose a language level solution for information flow control which assigns labels to every program object that incurs a substantial overhead. In both mechanisms, we have to change the source codes of the services. Therefore, we lose transparency which is a key factor in adoption of a technology by industry.

VI. Conclusion

In this paper we proposed a new end to end security auditing solution for SOA. The proposed solution is based on the introduction of two new security components, i.e. the *Taint Analysis* module and the *Trust Broker* service. By providing the ability to track external service invocations in the completion of a service request and maintaining dynamic trust values for services, the proposed architecture allows clients to be informed about the full chain of service invocations in a request and possible misbehavior by services involved in the request. This architecture both makes it possible to judge the quality of the response received by the client (i.e. judge the possibility of a tainted response) and increase the chances of selecting trustworthy services using the reputation based system.

The proposed end-to-end security auditing architecture is fully compatible with common Web services standards (WS-*), as the services and data communication protocol are not affected by the security related modifications (i.e. additions) in the general SOA structure. The minimal set of WS-* standards necessary to overcome the security challenges along with the proposed security components TA and *TB* were identified as WS-Security to ensure client and service authenticity as well as message level security through encryption and signing; and WS-Trust for the generation of security tokens required for authentication. By securing the communication between the taint analysis modules and the trust broker using WS-Security, the proposed system ensures authenticity of session feedbacks, hence preventing unfair increase/decrease of trust values of services due to targeted feedback from malicious parties.

Acknowledgment

We would like to thank Bala Gnanasekaran, Guneshi Wickramaarachchi, Ranjitkumar Sivakumar, Bill Pfeifer and Nwokedi Idika for their contributions to this project. This work was sponsored by Air Force Research Laboratory (AFRL), under award FA8750-10-2-0152 (104446). Any opinions expressed in this paper are those of the authors and do not necessarily reflect those of AFRL, or U.S. Government.

References

[1] J. Hutchinson, G. Kotonya, J. Walkerdine, P. Sawyer, G. Dobson, and V. Onditi, "Evolving existing systems to service-oriented architectures: Perspective and challenges," in *IEEE International Conference on Web Services (ICWS'07)*, 2007, pp. 896–903.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-oriented programming," *European Conference on Object-Oriented Programming (ECOOP'97)*, pp. 220–242, 1997.

[3] "JBoss AOP framework," http://www.jboss.org/jbossaop, [Online; Accessed Apr. 2012.].

[4] "AspectJ Framework," http://www.eclipse.org/aspectj/, [Online; Accessed Apr. 2012.].

[5] "AOP in Spring Framework," http://static.springsource.org/spring/docs/2.0.x/reference/aop.html, [Online; Accessed Apr. 2012.].

[6] "Web Service Specifications," http://en.wikipedia.org/wiki/List_of_web_service_specifications, [Online; Accessed Apr. 2012.].

[7] "WS-Security," http://en.wikipedia.org/wiki/WS-Security, [Online; Accessed Apr. 2012.].

[8] "WS-Security," http://goo.gl/CQ7BW, [Online; Accessed Apr. 2012.].

[9] "WS-Trust," http://en.wikipedia.org/wiki/WS-Trust, [Online; Accessed Apr. 2012.].

[10] "PicketLink Security Token Service," http://community.jboss.org/wiki/PicketLinkSecurityTokenService, [Online; Accessed Apr. 2012.].

[11] A. Ouda, D. Allison, and M. Capretz, "Security protocols in service-oriented architecture," in *6th World Congress on Services (SERVICES-1)*, 2010, pp. 185–186.

[12] M. Rahaman and A. Schaad, "Soap-based secure conversation and collaboration," in *IEEE International Conference on Web Services (ICWS'07)*, 2007, pp. 471–480.

[13] A. Benameur, F. Kadir, and S. Fenet, "Xml rewriting attacks: Existing solutions and their limitations," *Arxiv preprint arXiv:0812.4181*, 2008.

[14] "OASIS eXtensible Access Control Markup Language (XACML)," http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml, [Online; Accessed Apr. 2012.].

[15] R. Baird and R. Gamble, "Developing a security meta-language framework," in *Hawaii International Conference on System Sciences (HICSS 2011)*, 2011, pp. 1–10.

[16] R. Baird and R. F. Gamble, "Security controls applied to web service architectures," in *19th International Conference on Software Engineering and Data Engineering*, 2010.

[17] Z. Malik, "Rateweb: Reputation assessment for trust establishment among web services," *VLDB*, vol. 18, no. 4, pp. 885–911, 2009.

[18] G. Spanoudakis and S. LoPresti, "Web service trust: Towards a dynamic assessment framework," in *IEEE International Conference on Availability, Reliability and Security (ARES 2009)*, 2009, pp. 33–40.

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

[20] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006, pp. 19–19.

[21] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris, "Labels and event processes in the asbestos operating system," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 17–30, 2005.

[22] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 321–334.

[23] A. Sabelfeld and A. Myers, "Language-based information-flow security," *Selected Areas in Communications, IEEE Journal on*, vol. 21, no. 1, pp. 5–19, 2003.