An End-to-End Dynamic Trust Framework 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*—Service-oriented architecture (SOA) is an architectural paradigm that advocates composition of loosely-coupled services in order to construct more complex applications. The agility and complexity of modern web services on one hand and the arbitrary interconnections among them on the other hand, make it difficult to maintain a sustainable trustworthiness in long-running SOA-based applications. Moreover, the chain of participating services in a specific SOA invocation may not be visible to the service consumers, which leads to a lack of accountability. To address these challenges in SOA, we propose the following contributions. First, we design a new dynamic and flexible trust model based on graph abstraction that uses multiple trust strategies to calculate trust across SOA. This trust model keeps track of three trust metrics: individual service trust, session trust, and composite trust. We further design a trust engine component that implements the proposed trust model and that continuously maintains the quantitative end-to-end trust based on processing actual execution of services. Second, to prove the practicality and usefulness of the proposed framework, we have implemented an adaptive and secure service composition engine (ASSC) which takes advantage of an efficient algorithm to generate service compositions with near-optimal trustworthiness under predefined QoS constraints. Finally, we have developed a tool that is able to automatically deploy SOA testbeds from arbitrary directed acyclic graphs (created in the GUI). This tool enables the researcher to study the dynamics of new trust algorithms and strategies under different scenarios (e.g., arbitrary SOA topologies and attacks). We have extensively studied the effectiveness and performance of the proposed solutions using testbeds in the Amazon EC2 cloud.**

***Keywords*-Trust Management; Service-Oriented Architecture; Security**

I. Introduction

With the rapid growth of enterprise web-based services and the increasing popularity of cloud computing as an enabling technology, SOA (and similarly *microservices* architecture) is becoming a key software paradigm in designing large-scale distributed applications. SOA promises reusability, interoperability, and scalability of applications through chaining loosly-coupled services within or across organizational boundaries. Therefore, ensuring secure interactions among multiple services provided by different companies become a challenge. Majority of the current trust management systems are not designed for SOA and usually have wrong assumptions. For example, when a client contacts a front-end web service, the trust relationship is established between them through a public-key infrastructure (and certificate authorities). Therefore, regardless of whether this front-end service is going to respond back directly (a client-server model) or it is going to call a number of other services while it prepares the response (a SOA model), the whole transaction is considered as trusted for the client. This scenario, shows that there is an implicit assumption that trust is transitive. Therefore, once a client uses a service that she trusts, then she have to implicitly trust all other partners of that service. However, the trust is not necessarily transitive, specially when there are financial motives for using low-cost untrustworthy services. Another problem in the mentioned scenario is the fact that certificate-based trust model is static (as the certificates are usually valid for a very long time). However, the static and qualitative trust models are not useful for highly dynamic and agile SOA environments, where services change continuously and frequently. In fact, modern services change their internal implementations frequently without changing their interfaces, which makes it almost impossible for the service consumers to detect such changes. These changes can have a serious impact on trustworthiness of services. Finally, there is no mechanism for handling the trust in a chain of services in SOA.

In order to address the aforementioned challenges, we have proposed the following contributions:

1. A New trust model for SOA. We designed a new dynamic trust model for SOA that maintains the trustworthiness of services quantitatively. In designing this trust model, we show that the trustworthiness of SOA cannot be represented by a single metric. To address this challenge, we propose three trust metrics which represent different aspects of trustworthiness in SOA. These metrics differentiate the inherent trustworthiness of individual services from their interconnection and composite trust of a session of the whole SOA application. We further discuss three strategies in calculating composite trust. We also design an algorithm, called graph reduction trust algorithm, to efficiently calculate the composite trust in SOA. We have implemented this trust model as a software component called trust engine, which collects and processes a stream of security events generated by monitoring the execution of services. Trust engine also

makes the trust metrics accessible through REST APIs.

2. Adaptive and secure service composition engine (ASSC) for SOA. To prove extensibility and practicality of the proposed trust engine, we have designed and implemented a new policy-based service composition engine for SOA, which is able to adapt and reconfigure the service composition at runtime in order to maximize the trustworthiness of the overall SOA application. We adapt and use an efficient algorithm to find the most trustworthy service composition among available services in different categories, while meeting the cost and QoS constraints of the overall composition.

3. Automatic scenario generation tool. Finally, we have developed a flexible tool which is able to convert any directed acyclic graph into an actual SOA testbed (backed by interconnected RESTful web services). In order to build this system, we designed a generic web service which is configurable at runtime. This GUI-based tool simplifies studying the dynamics of multiple trust models and security measures under different scenarios (e.g., information disclosure, compromised service, and DoS attacks).

The rest of this paper proceeds as follows. In Section II we discuss the proposed architecture for end-to-end trust management in SOA. In section III, we design a new composite trust model for SOA, which is based on graph abstraction. Section IV describes the design of the adaptive and secure service composition engine (ASSC) for SOA. In section V, we discuss the experiments. Section VI covers the related work. We conclude in Section VII.

II. A High-level Overview of the Architecture

The main goal of this paper is to design a trust engine based on a new trust model for SOA that maintains multiple trust metrics. Trust engine is a software component which is designed as a part of a holistic security architecture for an end-to-end policy monitoring and enforcement in SOA. Further architectural details are available in [1]. In fact, the trust engine alongside with a few other components are parts of a trusted third party service called *Trust Manager* as illustrated in Figure 1.

The input of the trust engine comes from a collection of policy monitoring and enforcement (PME) components which are distributed across the SOA application. The PME components are responsible for monitoring the execution of services at runtime in order to detect and prevent malicious service invocations and illegal data leakage attacks. The internals of the PME components is discussed in [1]. In addition to trust engine that will be discussed in the next chapter, the Trust Manager has the following components:

Session management engine. This component is responsible for creating and maintaining a session for every request from service consumers. This session connects all relevant pieces of information collected from services that have participated in a specific service invocation. Further implementation details are discussed in [1].

Adaptive and Secure Service Composition (ASSC). ASSC component leverages an efficient algorithm to find the most trustworthy service composition among available services from multiple categories while meeting the predefined QoS constraints of the overall composition. This component is discussed in section IV.

Policy engine. The trust manager leverages an open source XACML policy engine (WSO2 Identity Server) to maintain the service consumers' policies. Policy engine is used to decide whether a certain operation (e.g., an external service invocation) made at a service is permitted or not. Once the trust manager gets the response of the policy engine, it respond back to the corresponding PME component to apply a certain enforcement strategy (e.g., block, permit, or redirect). Further details are discussed in [1].

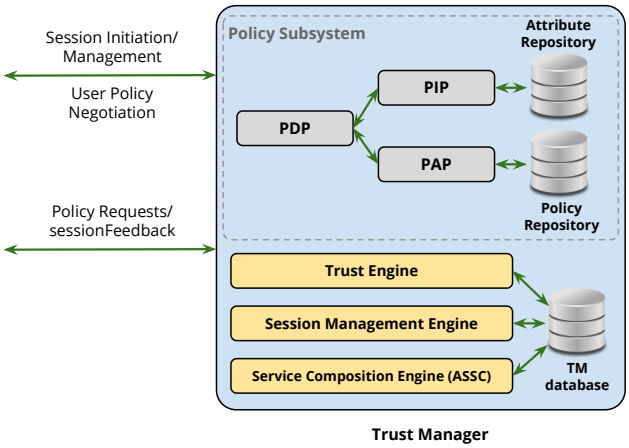

Figure 1. Architecture of the Trust Manager.

III. A Dynamic Trust Model for SOA

In this section we first define *trust* and *service graph*. Second, we discuss the basic topologies in SOA. Next, we present multiple trust strategies that could be used in calculation of composite trust. Finally, we present a graph-reduction trust algorithm, which gives a methodology to calculate end-to-end composite trusts.

A. Definitions.

We define trust as a measure that quantifies the likelihood of a service to perform as expected. Trust is represented as a real value from the range of $[0, 1]$, which 0 represent *not trustworthy* and 1 represents *fully trustworthy*.

Modern service-oriented applications require access to dynamic, quantifiable, and composite trust schemes that take the reputation and execution history of services into account. These trust schemes have to capture the interactions among services in SOA scenarios and calculate the trustworthiness of services based on service invocation topologies.

Due to complexity of SOA, a single trust value cannot represent the different aspects of trustworthiness. For example, Figure 2, shows at some point in the execution of a given SOA application, client C has invoked S_1 100 times. Likewise, S_1 has invoked S_2 99 times, and has invoked the S_3 just once. In such scenarios, considering the trust value of an individual service cannot be a good indicator for the trustworthiness of the whole service composition. To overcome this challenge, we maintain three types of trust values:

1. Service trust. This trust value indicates the inherent trustworthiness of an individual service which gets updated based on a trust update mechanism. We support four trust update mechanisms: Client's rating for the latest transaction (this rating is translated into a reward or punishment factor and will be applied to all participating services corresponding to that request); Session-trust based mechanism (which updates the trust values of the participating services based on whether trust manager considers the session as trustworthy or not according to the collected audit trail); Hybrid mechanism (which combines both of the previous mechanisms); Policy-based mechanism (which updates the service trusts based on a predefined policy and the current context).

2. Session trust. Session trust represents the trustworthiness of a single session. This metric is primarily used for reporting the trustworthiness of a specific service invocation (including all participating services in that request) to the client. For example, if in the last session of the application presented in Figure 2, S_1 has called S_2, then the session trust is calculated only based on the trust value of S_1 and S_2 but not S_3.

3. Composite (or aggregate) trust. This is a global trust value, which represents (predicts) the overall trustworthiness of a service composition initiated from a given service. This value can be considered as a prediction for the trustworthiness of a service based on its past history. For example, a client might want to know the expected trustworthiness of a service composition starting from S_1. In this case, we must take the execution history of all participating services (edge weights) into account.

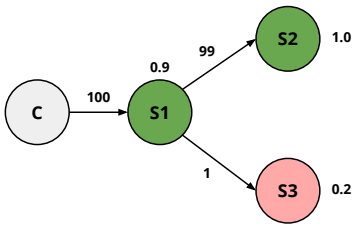

Figure 2. Example scenario for composite trust.

Therefore, session trust and composite trust depend on the following two factors:

- *Trustworthiness of all individual services.* It means the trust of a composite service is a function of its constituent sub-services.
- *Structural dependencies among those services.* It means the definition of trust in SOA must capture the way services are interacting with each other.

Definition of service graph (SG). A *service graph* is defined as a weighted directed acyclic graph (DAG), G, with an ordered pair $G = (V, A)$ of:

- V: which is a *set* of nodes. Each node represents a service in SOA. We associate a trust value to each node (service) in this graph.
- A: which is a *set of ordered pairs* of vertices, called *arcs*.[1] Each arc represents an invocation between two services and has a weight, which shows the number of invocations between these two services. For example, if S_i has invoked S_j 60 times in the past, the assigned weight to (S_i, S_j) is 60.

Trust manager is responsible for creating and maintaining the service graph, which captures the service interactions in a SOA system. This service graph is constructed over time based on the incoming session feedbacks—which include invoker and invoked services for every event—from the policy monitoring and enforcement (PME) components that monitor the services [1].

Definition of service composition graph (SCG). As mentioned, the definition of service graph is global and includes all services in a SOA system. We define *service composition graph* as a *subgraph* of a service graph. Whenever we query the trust manager for a composite trust of a service S_i, trust manager, creates a sub-graph of the service graph, which includes S_i and all other reachable services from this service.

B. Basic Topologies

Any arbitrary service graph is composed of a number of basic topologies. The basic topologies are used in the calculation of the composite trust based on a given trust strategy. The basic service interactions in SOA have the following forms:

1. Chained subcontract. In this topology, a service is only dependent on a single other service. In Figure 3 (a), S_i only requires to call S_j to perform its functionality. The weights on the arcs show that S_i is called n_i times in total by its predecessors and calls S_j for n_j times during the lifetime of service orchestration.

2. Conditional subcontract. In the second basic topological form, a service is conditionally depend on a number of other services. In this case, an invocation of dependent services does not happen for every service call. Therefore, each arcs shows how many times the target service is being

[1]In undirected graphs, these pairs of nodes are unordered and are called *edges*. Throughout this paper, we may use them interchangeably.

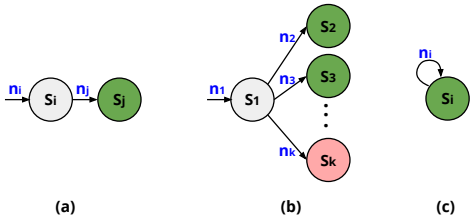

Figure 3. Basic topologies in SOA service graphs: (a) chained subcontract, (b) conditional subcontract, (c) recursive invocation

invoked so far. For example, in Figure 3 (b), S_1 is called n_1 times in total. During these n_1 calls, it has called each service S_i for n_i times ($i = 2, 3, ..., k$).

Note. The service graph is generated and maintained entirely based on empirical feedback data collected from PME components and we do not have any assumption on the internal logic of participating services. Therefore, we do not assume $\sum_{i=2}^{n} n_i = n_1$ as every time S_1 is called, it may call any subset of S_2, S_3, ..., S_k arbitrarily.

3. Self-invocation recursive call. This topology represents a service that recursively calls itself, which in graph theory they are called *self-loops*. Since in this case, no external service invocation happens, and the trust value of a service that invokes itself remains the same, we can safely ignore the self-loops.

C. Composite trust strategies

We have identified three main *strategies* that can be used to compute the composite service trust. No single strategy works best for all possible SOA scenarios. The choice must be made based on application's domain and sensitivity. These trust calculation strategies are as follows:

1. Pessimistic. The idea behind this strategy is the *principle of the weakest link* [2]. This principle states that the security of a system depends on the security of the weakest component in that system. Therefore, for any composite service composed of S_2, S_3, ..., S_n with an arbitrary topology, the composite trust would be calculated as follows:

$$t_{composition} = \min_{i=1..n} t_{s_i}$$

Even though this case is suitable for mission critical scenarios (e.g., military scenarios), however, it could be too restrictive for some commercial or less sensitive scenarios. For example, in service mashups, a web service may just invoke another service to get the latest weather service and display it on its page. In this scenario, weather service does not play a critical role in the overall functionality of the main service. If the weather service is untrusted, using the pessimistic strategy makes it a dominant factor in calculation of the composite trust, which is not reasonable.

2. Averaging. In this strategy, we define the composite trust as an average of the sub-services' trust values. Given a set of services $S_1, S_2, ..., S_n$, we define the composite trust as follows: $t_{composite} = \sum_{i=1}^{n}(t_{S_i})/n$.

3. Weighted Averaging. The intuition behind this case is that services have different impacts in a SOA application. For example, a popular service S_i that is invoked by more services should have a higher weight compared to its peers. Similarly, if a service S_i invokes a larger number of services, then it should have a higher weight, as the response of all invoked services will be returned to this service and it has a higher control over a larger subset of the service graph. Higher weight of a trusted service will make the service composition trustworthier and vice versa.

To formalize the weighting factor, we leverage the *PageRank algorithm* [3], which gives a higher weight to nodes with larger *indegrees*[2]. Applying the PageRank algorithm directly to the service graph does not reflect the second case mentioned earlier. To consider the effect of services with large fan-out, we augment the service graph by adding a reverse arcs for every arc in the service graph. This augmentation can be interpreted based on a fact that whenever a service S_i calls service S_j, S_j will reply back to S_i, which will be represented as a reverse arc. The output of applying the PageRank algorithm to an augmented service graph (for $S_1, S_2, ..., S_n$) is a list of scalar values $r_1, r_2, ..., r_n$ that $\sum_{i=1}^{n} r_i = 1.0$. The r_i values represent the weight of each service. Using these coefficients, we can calculate the composite trust of a graph using weighted averaging strategy as follows: $t_{composite} = \sum_{i=1}^{n} r_i.t_{S_i}$.

D. Graph-Reduction Trust Algorithm

The idea behind graph-reduction trust algorithm is the fact that every arbitrarily complex graph is composed of a set of basic topologies that are presented in Figure 3. Therefore, by replacing a basic topology in a graph by a node (service) with an equivalent trust value and weight, we can iteratively reduce a service composition graph to a single node with a trust value of the composite trust of the original graph. This algorithm is presented in Algorithm 1.

One of the main challenges in the graph-reduction trust algorithm is to find and calculate the basic topologies in a correct order. The intuition behind the proposed solution is to find a service (or services) in the composite graph that has no outgoing arc (which hereafter we call it a terminating service). By backtracking from a terminating service, we can identify a basic topology. This intuition can be implemented using topological sorting algorithm [4]. Since the service composition graphs are directed acyclic graphs, applying the topological sorting algorithm is guaranteed to produce a list of sorted services. After applying the trust algorithm to the identified basic topology, we remove all participating services in that basic graph from the sorted list and append

[2]*Indegree* of a node is the number of incoming arcs of that node. In the context of service graph, it shows how frequently a service is invoked by other services.

a new service with an equivalent weight and trust of the respective composite service.

In this algorithm, the averaging strategy is applied to the basic topologies, Figure 3 (a) and (b), as follows:
Topology (a): For basic topology (a), the composite trust is calculated as follows: $t_{composite} = \frac{n_i.t_{S_i}+n_j.t_{S_j}}{n_i+n_j}$. Similarly, the composite trust with weighted averaging strategy employs the PageRank weights as follows:
$t_{composite} = \frac{n_i.r_i.t_{S_i}+n_j.r_j.t_{S_j}}{n_i.r_i+n_j.r_j}$.
Topology (b): For basic topology (b), the weight of each service is used as the weight of the respected trust value. The trust value of the total topology is: $t_{composite} = \sum_{i=1}^{k}(n_i.t_{S_i})/\sum_{i=1}^{k} n_i$. The weighted strategy on topology (b) is calculated as follows:
$t_{composite} = \sum_{i=1}^{k}(n_i.r_i.t_{S_i})/(\sum_{i=1}^{k} r_i.n_i)$.

There is a non-trivial case (illustrated in Figure 4 (a)) that cannot be reduced to the basic topologies. This problem happens when the terminating service has $indegree > 1$. To resolve this issue, we duplicate the terminating service (with the same trust value and weight) and create $indegree - 1$ extra clones new of this service. Each of these services will be connected to the predecessor services through a single arc. The initial service in the sorted list will be replaced by new cloned services. After this processing, we can reduce the service graph using the basic topologies. Figure 4 (b) illustrates this operation. In this Figure, S_4 and S_5 have $indegree > 1$ and should be split. After applying the algorithm, S_4 and S_5 will be replaced by $S_4^{'}$, $S_4^{''}$, $S_5^{'}$, and $S_5^{''}$. The operation is presented in Algorithm 1.

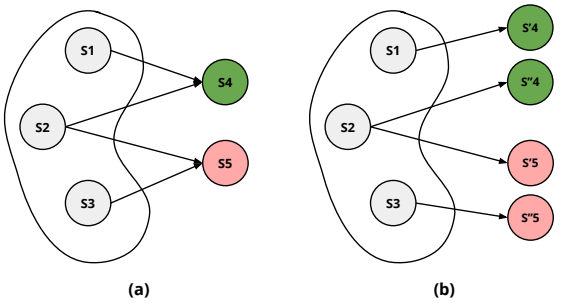

Figure 4. The problem of terminating services with $indegree > 1$: (a) problematic scenario, (b) a solution by cloning the terminating services.

The complexity of this algorithm is dominated by the BFS and topological sorting operations, which both of them have the complexity of $O(V + A)$. Therefore, the complexity of this algorithm is $O(V + A)$.

IV. An Application of the Proposed Trust Framework

To demonstrate the power of the proposed trust engine, we designed and implemented a policy-based service composition scheme (called ASSC) that uses the trust engine for adaptation and optimal service composition.

One of the benefits of SOA is the fact that it promotes an ecosystem, which encourages various companies to provide similar interoperable services. In this ecosystem, it is highly likely to find multiple equivalent services which provide similar functionalities. We define a *service category* as an abstraction for a set of services that have a similar service interface and provide a similar functionality. A *concrete*

Algorithm 1 The graph-reduction trust algorithm.

Input:
SG: The service graph $G = (V, A)$
S_{entry}: An entry service for the application
Output:
t_c: Composite trust of a subgraph with root S_{entry}.
Step 1: ▷ Extracting the SCG from SG.
    SCG ← subGraph(SG, S_{entry});
    ▷ $SCG = [S_1, S_2, ..., S_{n'}]$
Step 2: ▷ Topologically-sorting services
    L ← topologySort(SCG);
    ▷ $L = [S_1^{'}, S_2^{'}, ..., S_{n'}^{'}]$
Step 3:
    ▷ Calculating the composite trust based on strategy.
while $L \neq \emptyset$
    TS = L.getLast() ▷ TS: terminating service
    if $TS.indegree > 1$
        ▷ Figure 4 (b) case
        $L \leftarrow$ resolveByCloning(L);
        ▷ Updates the SCG; returns a new L
        ▷ $TOPO = \{S_1^{''}, S_2^{''}, ..., S_{n''}^{''}\}$ is the basic topology that encloses TS
    if strategy = *averaging*
        if ($TOPO$ is basic topology (a))
            ▷ In this case $TOPO = \{S_1^{''}, S_2^{''}\}$
            $t_c = (n_1.t_{S_1^{''}} + n_2.t_{S_2^{''}})/(n_1 + n_2)$
        else
            $t_c = \frac{(n_1.r_1.t_{S_1^{''}}+n_2.r_2.t_{S_2^{''}})}{(n_1.r_1+n_2.r_2)}$
    else if strategy = *weighted_averaging*
        if ($TOPO$ is basic topology (a))
            $t_c = \sum_{i=1}^{n^{''}}(n_i.t_{S_i^{''}})/\sum_{i=1}^{n^{''}} n_i$
        else
            $t_c = \sum_{i=1}^{n^{''}}(n_i.r_i.t_{S_i^{''}})/(\sum_{i=1}^{n^{''}} r_i.n_i)$
    else if strategy = *pessimistic*
        $t_c \leftarrow \min_{i=1..n''} t_{s_i^{''}}$
    replaceTopology($TOPO$);
    ▷ Replacing the $TOPO$ with a service, which has a trust value of t_c.
    $L \leftarrow L - TS$;
end-while
return t_c

service is a real implementation of that service category.

Since the modern services change frequently, client must be able to reconfigure and adapt the service compositions on-demand to achieve a higher composite trust. For example, once a service is attacked in a SOA scenario, two components of the trust manager (e.g., policy engine, and PME components) coordinate with each other to detect the policy violations. At this point, trust manager can trigger the ASSC component to find the current optimal service composition according to the latest SOA context (QoS and trust parameters of Services) and reconfigure the SOA application accordingly.

Since service consumers have different requirements (such as level of service, security assurance, etc.), service providers offer multiple services in the same category. In this business model, depending on the quality of the service and trustworthiness, they charge the service consumers with different rates. Moreover, competing service providers can also deliver similar services. The composition aims at selecting a set of services that provide the highest level of trustworthiness. However, there are some global constraints that must be met. For example, the total cost of all services in the composition, which should not exceed a predefined value C and the total end-to-end delay of all services should not exceed 300ms. In addition to these global constrains, we maintain an arbitrary number of per-service parameters.

ASSC component formulates the service composition problem as an instance of a *multiple-choice multi-dimensional knapsack* (MMK) problem. MMK problem is also an NP-hard problem [5]. We use M-HEU [6] that is a very efficient heuristic algorithm to solve the MMK problem. M-HEU algorithm is an optimized variation of HEU algorithm [7]. Akbar et al. [6] proposed an incremental heuristic algorithm to solve the multidimensional knapsack problem. We have implemented with slight modifications. To gain a faster (and time-bounded) solution, we modified the M-HEU by limiting the number of upgrades and the number of downgrades at the expense of losing a negligible optimality of a solution. We further improved the M-HEU by adding a preprocessing step and also improving the selection of the initial feasible solution. The improved algorithm is implemented as a generic algorithm, which can operate on an arbitrary number of services and QoS constraints. The details of the formulation of this algorithms is available in chapter 6 of [8]. We implemented the ASSC as a web service inside the trust manager service, which is accessible through four REST APIs.

The QoS constrains (e.g., response time) is implemented through an aspect which is part of the PME component and instruments the Jersey REST framework to collect the execution time of the POST calls. Once this pointcut is matched, the corresponding advice is called to calculate the response time of the target service and store that in the trust manager.

V. Implementation and Evaluation

A. A tool for automatic generation of arbitrary SOA testbeds

As a part of this project, we developed a GUI to simplify the interaction of clients with the security framework. Moreover, we have designed a new tool for this GUI that enables the researchers to create, deploy, and study any arbitrary SOA scenario with a few clicks. A screenshot of this GUI is shown in Figure 5. This tools captures the graphical model of the target SOA application as a directed acyclic graph and performs a series of analysis to check the validity of this model (e.g., being loop-free) and extract the interconnection among those nodes. We have developed a very versatile web service called *relay service*, which exposes REST APIs for its internal reconfiguration. For example, we can configure which services it should invoke per incoming requests. We also can remotely enable an attack (information disclosure or DoS attack) on this service. The proposed tool deploys a sufficient number of these relay services and configures them to create the target SOA scenario.

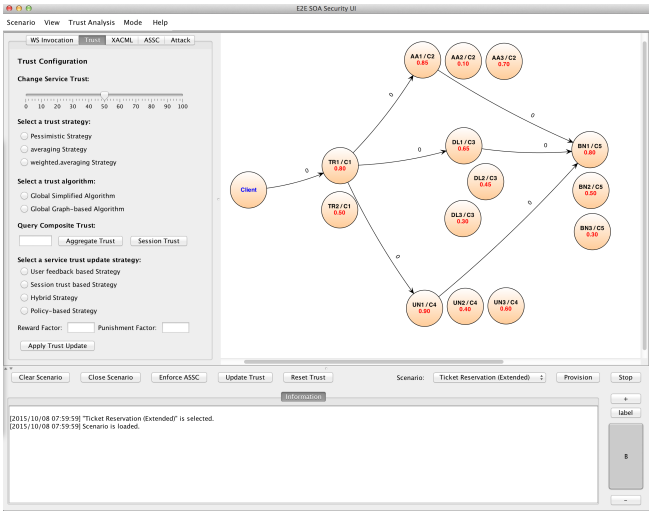

Figure 5. A screenshot of the user interface during the ticket reservation scenario.

Using this UI, we can upload, enable, or disable arbitrary XACML policies at runtime. Moreover, it makes all of the discussed trust parameters, strategies, and update mechanisms accessible and configurable.

B. Experiments and Results

The goal of this experiment is to verify the efficiency of the ASSC component and the trust engine in term of response time. For this experiment, we deploy the testbeds in the cloud. All experiments are conducted in the Amazon EC2 cloud. In these experiments, we deployed the application services, policy engine, and trust manager in t2.small instances (1 vCPU, 2GB memory, and EBS storage). To minimize the effect of unpredictable and variable delays

on the client side, the client (Apache Benchmark[3]) is deployed in a separate t2.micro instance. Each experiment is repeated 5000 times and the results are the average of these experimental runs. There are three input parameters that we will investigate their effects on the performance of the ASSC component. These parameters are the total number of participating services, number of service categories, and the number of composition constraints.

At first, we investigate the effect of the number of services on the response-time performance of the ASSC. These performance values include the response time of the trust engine component too. The other two parameters are set to 5 for the number of categories and 3 for the number of composition constraints. The response time of the ASSC REST API for scenarios with total number of services from 25 to 125 is presented in Figure 6. The results show that the execution time changes almost linearly. Even for 125 services in 5 categories (which is unlikely to be surpassed in any practical SOA scenario), the ASSC service performs very well and the average response time is 22ms.

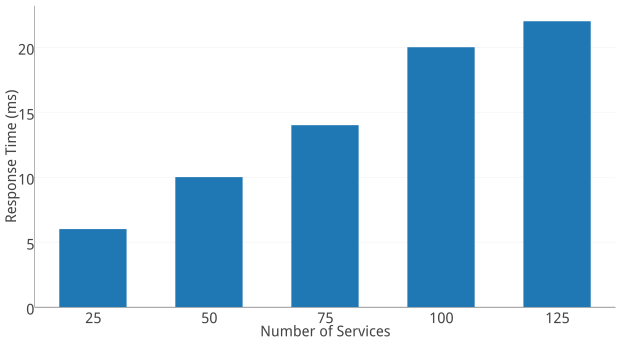

Figure 6. Response-time of the ASSC component in the Amazon EC2 testbed 1 with 5 service categories and 3 constraints.

In the second experiment, we investigated the effect of the number of service constraints on the performance of ASSC component. In this experiment, we have set the number of services to 50 and the number of categories to 5. According to Figure 7, the effect of the QoS constraints on the performance of ASSC component is sub-linear. Even after increasing the input size by a factor of 5, the response time only increases 50%.

In the third experiment, we investigate the effect of the number of service categories on the performance of ASSC component. In this experiment, the number of services is set to 50 and set the number of constraints is set to 3. Similar to the previous results, based on Figure 8, the ASSC response times are fast and they do not exceed 20ms for 25 categories.

The discussed performance evaluation experiments confirm that the ASSC and the trust engine components are responsive and efficient for any practical SOA application.

[3]https://httpd.apache.org/docs/2.4/programs/ab.html

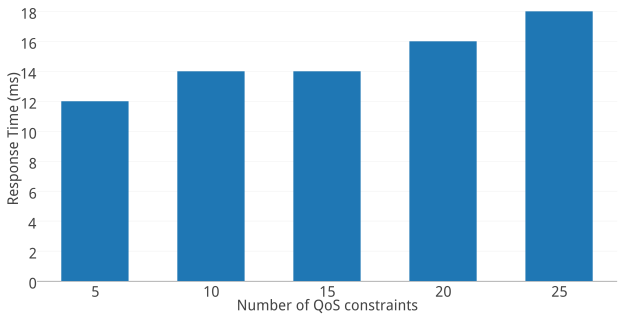

Figure 7. Response-time of the ASSC component in the Amazon EC2 testbed 1 with 50 services and 5 service categories.

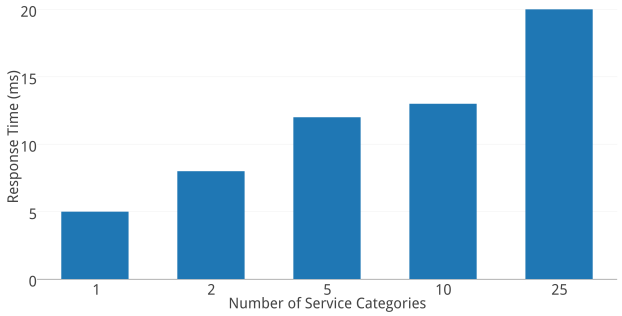

Figure 8. Response-time of the ASSC component in the Amazon EC2 testbed 1 with 50 services and 3 service constraints.

VI. Related Work

Trust models have been extensively studied in the literature under different contexts [9], [10]. However, majority of them are based on mathematical modeling (statistical or game theory models), without going beyond simulation studies. Furthermore, majority of these models are based on reputation-based trust. However, in this paper, we have provided a realistic and flexible trust model with multiple trust strategies, which is based on collecting actual evidences from the execution of services.

Malik [11] introduces a peer-to-peer framework for a trust-based service selection and composition, which is based on reputations of services made by their peers. Even though this approach is more resilient to failures, however, it is less efficient for large-scale deployment as every service requires to collect information about other services. Can and Bhargava [12] proposed a similar self-organizing peer-to-peer trust model. In this model peers create their own trust network using locally available information. Trustworthiness of peers in a network is evaluated using two contexts, services and recommendation of contexts.

Service composition in web services and SOA is a challenging research area that has attracted many researchers in the recent years. Dustdar et al. provides a comprehensive survey of web service composition techniques in [13]. They categorize the service composition techniques into static, dynamic, declarative, and model-driven. In the static service composition, service selection happens once at the deployment time and remains fixed during the lifetime of its execution. Our proposed service composition falls into the dynamic category.

Authors in [14] present a formulation of multiple knapsack problem for web service composition. However, they only discuss the problem in the context of QoS and do not address the security criteria. Moreover, they do not provide any real-world experimental study. Hang and Singh [15] propose a trust-based composition approach, which uses Bayesian networks. However, their proposed approaches are highly theoretical and they only provide a simulation study to verify their effectiveness. Authors of [16] has proposed a similar approach based on Mixed Integer Program, which similar to the previous approach, they have focused on QoS parameters.

VII. Conclusion and Future Work

In this paper, we first presented a new dynamic trust model for SOA which is able to capture the multiple aspects of the trust in SOA. Next, we presented an adaptive and secure service composition engine which takes advantages of the proposed security model and proves the practicality and usefulness of this model. Third, we provided a tool which is able to deploy arbitrary SOA scenarios from an input graphs. Finally, we evaluated the performance of the ASSC engine in the EC2 cloud environment to evaluate the performance of the proposed components. All experimental results verify the performance of the proposed components. The response times of the ASSC component under all scenarios remain less than 22ms.

There are several directions to extend the current work. One future direction would be to use the proposed tools in this paper to study the performance of multiple trust models and strategies under different service topologies. Another future work, would be to integrate the proposed trust models into existing open source SOA engines.

References

[1] M. Azarmi and B. Bhargava, "End-to-end policy monitoring and enforcement for service-oriented architecture," preprint. Available at: http://mazarmi.org/doc/pme.pdf.

[2] C. Flink, "Weakest link in information system security," in *Workshop for Application of Engineering Principles to System Security Design (WAEPSSD)*, 2002.

[3] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web." 1999.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and Stein, *Introduction To Algorithms*, 3rd ed. MIT press Cambridge, 2009.

[5] M. Hifi, M. Michrafy, and A. Sbihi, "Heuristic algorithms for the multiple-choice multidimensional knapsack problem," *Journal of the Operational Research Society*, vol. 55, no. 12, pp. 1323–1332, 2004.

[6] M. M. Akbar, E. G. Manning, G. C. Shoja, and S. Khan, "Heuristic solutions for the multiple-choice multi-dimension knapsack problem," in *Computational Science, (ICCS)*. Springer, 2001, pp. 659–668.

[7] S. Khan, "Quality adaptation in a multi-session adaptive multimedia system: Model and architecture," *Canada: Department of Electronical and Computer Engineering, University of Victoria. Thesis (PhD)*, 1998.

[8] M. Azarmi, "End-to-end security in service-oriented architecture," Ph.D. dissertation, Purdue Univeristy, 2016.

[9] O. A. Wahab, J. Bentahar, H. Otrok, and A. Mourad, "A survey on trust and reputation models for web services: Single, composite, and communities," *Decision Support Systems*, vol. 74, pp. 121–134, 2015.

[10] W. Viriyasitavat and A. Martin, "A survey of trust in workflows and relevant contexts," *Communications Surveys & Tutorials, IEEE*, vol. 14, no. 3, pp. 911–940, 2012.

[11] Z. Malik and A. Bouguettaya, "RATEWeb: Reputation assessment for trust establishment among web services," *The International Journal on Very Large Data Bases*, vol. 18, no. 4, pp. 885–911, 2009.

[12] A. B. Can and B. Bhargava, "SORT: A self-organizing trust model for peer-to-peer systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 1, pp. 14–27, 2013.

[13] S. Dustdar and W. Schreiner, "A survey on web services composition," *International Journal of Web and Grid Services*, vol. 1, no. 1, pp. 1–30, 2005.

[14] T. Yu and K.-J. Lin, "Service selection algorithms for composing complex services with multiple QoS constraints," in *Service-Oriented Computing-ICSOC 2005*. Springer, 2005, pp. 130–143.

[15] C.-W. Hang and M. P. Singh, "Trustworthy service selection and composition," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 6, no. 1, p. 5, 2011.

[16] M. Alrifai, T. Risse, and W. Nejdl, "A hybrid approach for efficient web service composition with end-to-end qos constraints," *ACM Transactions on the Web (TWEB)*, vol. 6, no. 2, p. 7, 2012.