Guarding Sensitive Information Streams through the Jungle of Composite Web Services

Jinpeng Wei, Lenin Singaravelu, Calton Pu
Georgia Institute of Technology, Atlanta, GA, USA
{weijp,lenin,calton}@cc.gatech.edu

Abstract

Complex and dynamic web service compositions may introduce unpredictable and unintentional sharing of security-sensitive data (e.g., credit card numbers) as well as unexpected vulnerabilities that cause information leak. This paper describes a fine-grain access policy specification of security-sensitive data items for each component web service. We propose the SF-Guard architecture to enforce these access policies at component web services. A prototype implementation of SF-Guard (on Apache Axis2) and its evaluation show that effective protection of security-sensitive information can be achieved at low overhead (a few percent addition to response time) while preserving the functionality of flexible web service composition.

1. Introduction

Web services enable composition of loosely coupled components into sophisticated applications [10]. As an example, consider the Travel Agent web service shown in Figure 1. Each component may represent an independent company which not only exposes a set of web service interfaces, but also relies on web services provided by other companies. While these composite web services are very useful and increasingly popular, they raise serious concerns about the protection of security-sensitive information such as credit card and social security numbers. Current web service standards such as WS-Security [20] and Platform for Privacy Preferences (P3P) Specification [16] have been defined to protect information exchange between a client and server, but offer limited or no support for composite services. For example, both WS-Security and P3P assume that each web service node is trusted to handle all user data, including the security-sensitive information.

There are several reasons for avoiding the current model of completely trusting component web services. For example, a legitimate service may have been infiltrated by malware capable of stealing security-sensitive information. Another possibility is the acquisition of a service provider by another company that may use previously collected information under different privacy policies.

Compared to Mandatory Access Control [4] for *centralized* operating systems [11][8], a composite web service environment has the following properties that make it different from traditional access control protection mechanisms:

Property 1: Decentralized authority of each component web service. As expected in web service environments, each component has its own protection boundaries.

Property 2: Multiple namespace managers. In a composite web service, node and component identities need to be consistently understood by participating nodes.

Property 3: Isolation of sensitive information from the intermediate web service nodes that should not have access.

The first contribution of the paper is the SF-Guard architecture, which supports a fine-grain policy-based access control model to control and protect propagation of security-sensitive information through multiple component services. We translate a user's security and privacy requirements into a set of access control policies, which are encoded into a *security-policy envelope* (SPE) that encapsulates security-sensitive data. These SPEs enforce appropriate access to encapsulated user data by each component web service as defined by the security policies.

The second contribution of this paper is a concrete demonstration of SF-Guard architecture, consisting of an API called WS-SensFlow and a prototype implementation. The current version of WS-SensFlow focuses on fine-grained access control. The prototype implementation of SF-Guard is called SG-Wrapper (built on Axis2 framework and toolkit) that performs the following functions: (1) intercept incoming invocations and replace sensitive data with capabilities; (2) carry out operations on the sensitive data on behalf of

the web service routines; (3) intercept outgoing invocations to ensure that sensitive user data is not leaked in unwanted ways. An experimental evaluation of SG-Wrapper shows strong protection properties and low overhead.

The rest of this paper is organized as follows. Section 2 presents the WS-SensFlow API and the specification and generation of SPEs. Section 3 describes the SF-Guard architecture and its wrapper-style design on each web service node. The Axis2-based implementation of SG-Wrapper is outlined in Section 4 and its evaluation is discussed in Section 5. We talk about related work in Section 6 and conclude in Section 7.

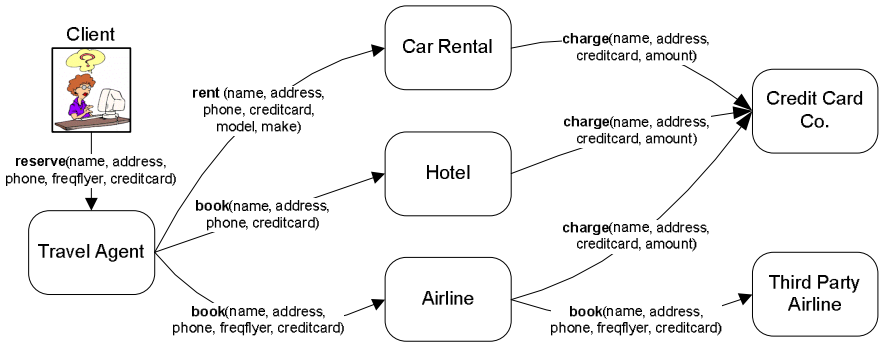

Figure 1: Travel agent composite web service

2. WS-SensFlow

WS-SensFlow is an API for fine-grain, policy-based protection of security-sensitive information propagation through multiple composite web services. It is policy-based because it allows the specification and attachment of security policies to the web service invocation requests. It is fine-grain because different policies can be specified for different input data items within the same invocation. These policy specifications are guarded by the SG-Wrapper on participating web service nodes (Section 3).

2.1. Security Policies

We regard sensitive information as a special resource, to which accesses should be controlled. Thus we reduce the information protection problem into an access control problem. The *subjects* here are the web service nodes, and the *objects* are sensitive information. Due to the open and distributed nature of our problem domain, a complete access control matrix cannot possibly be created. Fortunately, this is not necessary because each user only needs to specify which subjects can or cannot access her data. So we define security policies in WS-SensFlow as access control lists (ACLs), which are encoded into Security Policy Envelopes (SPEs). Specifically, a SPE ***L*** lists the web service nodes that are allowed to access a data item (*white list*), and the web service nodes that are not (*black list*).

L = <*white list*>; <*black list*>
<*white list*> = **allow** <*node list*>
<*black list*> = **deny** <*node list*>
<*node list*> = * | <*node id*> | <*node id*>, <*node list*>

The asterisk (*) is a notation that represents all participating web service nodes.

We assume that there is a way of uniquely mapping *node id* to a web service node, e.g., URI or public key. The list of web service nodes involved in a composition can be discovered using the tool described in Section 4.1.

Discussion: We can also use the traditional format of <*category, level*> [8] for the SPEs. The problem is that receive levels (clearances) need to be assigned for each service node before invocation. The ACL-style SPE removes such a configuration phase, so that each invocation message is self-contained – each intermediate web service node only needs to look at the SPE to make a decision whether to forward a data item to the next web service node or not.

Our current policy specification of white lists and black lists is a simple one. More complex policies, e.g., a policy with time attributes, can also be incorporated in WS-SensFlow with modifications to the enforcement mechanism (described in Section 3.3.3).

Running Example. In the Travel Agent application (Figure 1), a client needs to buy a plane ticket, book a hotel room, and rent a car. She invokes the service provided by a travel agent. She tells the travel agent her name, address, phone number, frequent flyer number and credit card number. The travel agent uses such information to invoke the Airline, the Hotel and the Car Rental web services on behalf of the client. Each of the latter three web services in turn invokes a Credit Card Company to charge the client. Occasionally the Airline rebooks tickets from a Third Party Airline. Note that this last invocation does not always happen, but it may be undesirable to the client because she does not want her frequent flyer number to be released to the Third Party Airline. To prevent this kind of situation, the client-side application can attach the SPE L_1 = {**allow** Travel Agent, Airline, Hotel, Car Rental, Credit Card Co.; **deny** Third Party Airline} to the frequent flyer number in the invocation message. Then the basic access control rule guarantees that the

client's frequent flyer number can be propagated to all web services in Figure 1 except the Third Party Airline web service. However, the client may specify the SPE L_2 = {**allow** *} for her name in the same invocation message, because she does not care if the Third Party Airline knows her name.

2.2. Security Policy Specifications

Due to the loosely-coupled nature of composite web services, a participating node can change its implementation by invoking different external web services without notifying its callers. As a result, the set of web service nodes participated in an invocation (composite service topology) can change from time to time. Even if the composite service topology remains static, the amount of trust put on each of the web service nodes by a client can change (e.g., through experience or recommendations). Therefore, WS-SensFlow supports dynamic service policy specifications.

A security policy specification in WS-SensFlow is a function from a set of web service nodes to the set {**allow, deny**}. In other words, a security policy specification maps a web service node to either the white list or the black list.

This specification process can be refined into two orthogonal sub-processes: (1) find out the set of participating web service nodes in a composite web service (we refer to this set as **W**), and (2) decide the mapping.

2.2.1. Composite Service Topology Discovery. Finding out W is important because otherwise all web service nodes (whether participating in the composite service or not) will be the potential recipients of sensitive data, which leads to huge white and black lists (SPEs). On the other hand, the knowledge about W can give much better idea about which nodes should be considered and result in much smaller white and black lists.

WS-SensFlow requires that each web service node provide the following meta-information:

- A URI which identifies the web service node;
- A list of the service routines provided and implemented by this web service node;
- A list of external service routines (including the URI of the provider) invoked by this service node.

Moreover, to support correlation of different web services, e.g., to find out that Travel Agent passed on **phone** and **creditcard** information from the client to the Hotel, WS-SensFlow requires that such service routine description on the various nodes support consistent meaning for the input parameters (e.g., **phone** means that the corresponding parameter is interpreted as a phone number across all web service nodes). Web Ontology [15] can be used for such purposes.

Using such meta-information, the complete, nested invocation relationship of a composite web service (e.g., the one in Figure 1) can be computed, which helps answer the question of which web service nodes can potentially receive a security-sensitive data item. The collecting of such meta-information can be done either statically (e.g., at development time), or dynamically (e.g., at invocation time). We have implemented a dynamic topology discovery tool which will be described in section 4.1.

2.2.2. Generation of SPEs. The second sub-process of security policy specification is to divide a set of web service nodes into a white list and a black list. Exactly which node goes to which list is application specific and therefore beyond the scope of WS-SensFlow. However, WS-SensFlow offers the following guideline regarding how this can be done.

Based on the application's knowledge about the set of web service nodes, they can be classified into three groups: nodes that are trusted, nodes that are not trusted, and nodes that are not yet known well enough to make a judgment. The application can set up a white list for web service nodes that are trusted, and a black list for web service nodes that are not trusted (Section 2.1).

For the unfamiliar nodes the application designer can leverage on the extensive amount of research on reputation systems (such as [25] and [6]). E.g., web service nodes with good reputation should be put in the white list, while others should be put in the black list. The application designer can also make use of an existing trust service such as WebTrust [24]. Therefore we assume that there is an agent which can answer queries about a web service node's reputation.

Once the white list and the black list for a data item (e.g., frequent flyer number) are constructed, a SPE (Section 2.1) can be generated and attached to the data item in the invocation message (e.g. SOAP). Such SPEs will be used on a participating web service node to decide how to treat the corresponding data item.

2.2.3. Ease of Security Policy Specification. WS-SensFlow will be less useful if it incurs unduly burden on the end user in terms of specifying the security policies for each web service invocation. Therefore WS-SensFlow separates three kinds of people who can specify policies:

- Application designers who can embed common security policies at compile time (e.g., for nodes that are well-known to be good or bad). Application designers can also embed calls to reputation systems to dynamically categorize component nodes.

- System administrators who can define or update security policies at setup time.
- Finally, end users who can define or override the default security policies at run time.

The amount of specification effort is assumed to be the most at the compile time, less at the setup time and the least at run time. The goal is to minimize the effort of the end user but reserve the rights of the end user to specify her own security policies.

3. SF-Guard Architecture

In our architecture, each web service node will have a SG-Wrapper that is responsible for enforcing and propagating SPEs.

3.1. Threat Model

Our architecture makes the following assumptions: (1) there is a Trusted Computing Base (TCB) [7] on each web service node. This TCB includes the hardware, the operating system, the web service supporting middleware (e.g., Java Virtual Machine and Web Service Framework), and SF-Guard. (2) The web service routines (business logic) are untrusted.

Under the circumstance of this paper, (2) means that a service routine can intentionally *leak* the security-sensitive information to some unwanted subjects. For example, the Airline web service routine in Figure 1 can leak the client's frequent flyer number to the Third Party Airline web service node in various ways in addition to normal web service invocation: storing it onto a removable disk and then copying it to the Third Party Airline machine, or sending it directly to Third Party Airline machine via FTP (File Transfer Protocol). The Airline web service routine can also transform the frequent flyer number in arbitrary ways before propagating it to the Third Party Airline, which then recovers it. Finally, the Airline web service routine can leak information to some other arbitrary machines not shown in Figure 1, using the above-mentioned ways. Such leakage equals granting access to the security-sensitive information to subjects unexpected by the end user, which may defeat the use of SPEs.

This situation reminds us of the confinement problem [12], which prevents a program from transmitting information to any other program except its caller. One example confinement technique is sandboxing, which restricts the access of a confined program to disk, network or other output channels.

Confinement is easier if accesses to such *legitimate* channels can be completely disabled. However, many a time a web service routine needs to access these channels to fulfill its normal task. Then in order to confine the untrusted service routine we must be able to mediate the access requests and check that only permitted data is output. However, intercepting every output request and checking every output data item is non-trivial. Besides, even if this can be done, the untrusted web service routine can still exploit *covert* channels [12] to leak information.

Thus we address this problem from a different angle: instead of giving the security-sensitive information to the untrusted web service routine and then trying to confine the routine, we use capability-based access control [13] to hide the sensitive information from the web service routine in the first place, thus avoiding the needs for confinement. The details and the justification for this design are discussed below.

3.2. Capability-Based Protection

In current web service middleware, the data exchange format between the underlying framework and the business logic (e.g. web service routines) is XML. Request/response data (SOAP message body) is directly given to the business logic, and it is up to the business logic to parse and interpret the SOAP message. While this is a reasonable design (because the underlying framework can not know the meaning of every kind of SOAP message body), it poses difficulties for confinement, because sensitive information, if there is any, is always exposed to the business logic.

To address the problem of over-trusting the business logic, we employ a capability-based access control on the sensitive data. We add the *SG-Wrapper* (Figure 2) between the web service framework and the business logic to hide the sensitive information from the latter. Specifically, sensitive data is extracted from the SOAP body and replaced by unique, non-forgeable *capabilities* before it is delivered to the business logic. Afterwards the business logic can access the sensitive data only through pre-defined interfaces. E.g., when the business logic needs access to the sensitive data, it presents the capability and calls the pre-defined interfaces. In this way, the business logic does not see the actual sensitive information. We put SG-Wrapper into the TCB.

To make sure that capabilities can not be forged or tampered with by the business logic, we can encrypt an internal counter and use the result as capabilities, and the significant bits of a capability should be large enough (e.g., 128 as in Amoeba [1]).

Capability-based access control is suitable for encapsulating sensitive information in our problem domain because of the following observations.

- Such sensitive information does not need complex computation. For example, it makes no sense to carry out arithmetic operations on social security number or a person's religion. In particular, such sensitive information is read only.

- Such sensitive information is a kind of atomic object whose meaning will be lost or distorted if not presented as a whole. For example, an individual digit of a credit card number is not a secret, but putting all the digits together in a particular order is.

Therefore, we assume that sensitive information should be read-only and presented in entirety. This enables us to encapsulate any of such sensitive information into an object with a few pre-defined interfaces. Following the two observations above, we only need output interfaces (such as displaying, printing, writing to a file, or sending out to the network).

However, although this design prevents direct access to the sensitive information, it does not necessarily prevent indirect accesses. For example, the business logic can request the data to be written into a file that it can read later. So SG-Wrapper must perform proper *declassification* of the sensitive information in such cases. For example, several digits of a social security number can be masked off before it is written to a file.

3.3. The Wrapper-Style Design

We implement SF-Guard by adding SG-Wrapper on each web service node to hide sensitive information and enforce SPEs. SG-Wrapper is part of the TCB (Trusted Computing Base) on the web service node. TCB is required in this framework to make sure that SG-Wrapper cannot be bypassed.

In detail, SG-Wrapper maintains a secure object repository which holds the sensitive information. Each secure object is instantiated from a sensitive data item in the incoming SOAP message. A secure object also provides a set of interfaces for outputting the sensitive data.

As mentioned above, the secure objects are used to conceal the sensitive information from the untrusted business logic, such that the latter can only refer to the sensitive information using capabilities. So SG-Wrapper needs to maintain a mapping from the capabilities to the secure objects.

3.3.1. Incoming Message Sanitization. When SG-Wrapper receives an incoming SOAP message, it transforms every data item with a SPE (section 2.1) in the following steps:

- Extract the data item from the message, and create a secure object for it. The SPE is also stored in the secure object.
- Replace the original data item with the capability associated with the corresponding secure object.
- Pass the sanitized SOAP message up to the business logic.

3.3.2. Normal Operations on Sensitive Information by the Business Logic. During execution, the business logic can access a sensitive data item only through SG-Wrapper by using its capability. That is, the business logic invokes the pre-defined interfaces provided by SG-Wrapper, and SG-Wrapper carries out the operation on behalf of the business logic. Based on the observations in Section 3.2, the set of pre-defined interfaces should be enough to satisfy the business logic's needs.

This design enables us to add different policies in terms of how the business logic can access the sensitive information. For example, we can deny a request to dump the sensitive information into a publicly accessible file.

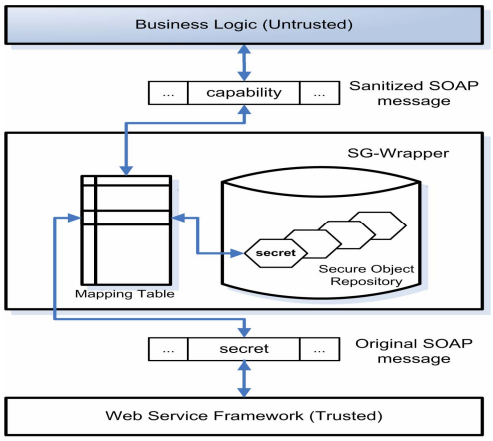

Figure 2: SG-Wrapper structure

3.3.3. Outgoing Message Processing. When the business logic needs to invoke an external web service, it forms a SOAP message which is intercepted by SG-Wrapper. If any sensitive information is needed, the business logic refers to it using a capability in the SOAP message.

SG-Wrapper does the following things for each capability in the message:

1. Map the capability to a secure object in the repository.
2. Fetch the SPE from the secure object.
3. Apply the security policy associated with the secure object. In this case, match the destination of the SOAP message against the SPE. The outgoing SOAP request will be rejected if the destination is on the black list, or if it is not on the white list (if the white list is "**allow** *", the destination is considered to be on the white list). Black list always takes precedence in making the decision. To enforce richer policies, SG-Wrapper has to be modified appropriately.
4. If the destination is on the white list and not on the black list, fetch the original sensitive information as well as the SPE from the secure object and put them into the outgoing SOAP message.

If the SPEs of all capabilities allow the destination, the SOAP message is passed on to the next mod-

ule (e.g., WS-Security) for further processing. Otherwise the output SOAP request is denied.

4. Prototype Implementation

We used the Axis2 web service framework [3] to implement SF-Guard. In the following we first describe the implementation of a dynamic topology discovery tool in Section 4.1. Then we discuss the implementation of SG-Wrapper in Section 4.2.

4.1. Dynamic Discovery of Composite Service Topology

As mentioned in Section 2.2.1, we designed and implemented a tool to dynamically compute the composite service topology, which is very helpful in the process of security policy specification. This tool is implemented as a web service routine on each web service node.

A composite web service may in turn invoke other web services, so we represent the web service invocation topology as a call graph with the web services as nodes and the invocations as edges. The goal of this tool is to build this call graph. This computation is a recursive process starting from the *root* web service node, which offers the composite web service. Each web service node collects information from its descendents – the web services that it invokes, adds its own information and sends the final results to its predecessor.

The core of our topology discovery tool is a distributed algorithm called **GetTopo**, whose pseudo code is shown in Figure 3. **GetTopo** uses two pieces of local information: the names (URIs) of the web service nodes that a service node invokes (*Callees*), and the name (URI) of a service node itself (*Name*). We assume that information about *Callees* is available on each web service node.

The main body of **GetTopo** is derived from a distributed depth firth graph traversal algorithm. Since a call graph may contain loops, the input parameter *visited_sofar* is used to convey the list of service nodes already visited to the next service node, so that it will not call **GetTopo** on these nodes again. Similarly, when **GetTopo** returns, the list of service nodes that have been visited is derived from the results by calling **NodesIn**, and added to the list of service nodes already visited.

4.2. Implementation of SG-Wrapper

SG-Wrapper (Section 3.3) is implemented as an Axis2 module [2], which intercepts both incoming and outgoing SOAP messages (Figure 2). In a SOAP message body, each parameter is represented as an XML element. SG-Wrapper works by checking and manipulating XML attributes in such XML elements. For example, the attribute named '*whitelist*' gives the list of web service nodes that can access the value of the corresponding input parameter (Section 2.1). Similarly, when a sensitive parameter is replaced with a capability (Section 3.3.1), an XML attribute called '*capability*' is inserted to the corresponding XML element. This attribute lets the business logic know that the corresponding parameter is just a capability, not the true input data.

When a sensitive parameter is replaced with a capability, a wrapper object for the Secure Object Repository is passed on to the business logic through the Axis2 message context [2], which is shared between SG-Wrapper and the business logic. The business logic then obtains this wrapper object and uses Java reflection API [19] to call the pre-defined interfaces (Section 3.3.2) if it needs to.

```
Set of String Callees;
String Name;
Set of Pairs GetTopo (Set of Strings visited_sofar) {
  WSNode N;
  Set of Pairs result = empty;
  Set of Strings visited = visited_sofar ∪ {Name};
  for (each C in Callees){
    result = result ∪ {<Name, C>};
    if (C ∉ visited){
       N = GetWSNodebyName(C);
       newpairs = N.GetTopo(visited);
       result = result ∪ newpairs;
       visited = visited ∪ C;
       visited = visited ∪ NodesIn(newpairs);
    }
  }
  return result;
}
```

Figure 3: Pseudo code of GetTopo on each web service node

5. Evaluation of SF-Guard

5.1. Experiment Setup

In order to evaluate SF-Guard, we implemented the 6 web services shown in Figure 1. We assume that the decision process (e.g., comparing prices and choosing a hotel) has been done, and the client application just wants to finalize the reservation. To do that, the client application first invokes the **getTopo** web service of Travel Agent to learn about the participating web service nodes. Then the client application attaches to the user's information (e.g., credit card number and frequent flyer number) the appropriate SPEs (Section 2.1), which in turn get translated into XML attributes in the outgoing SOAP message. This finishes the bootstrapping process.

To provide end-to-end protection of sensitive information, we also applied WS-Security [17] on each

web service node. Besides, each web service is run on a dedicated host and synchronous web service invocations are used.

5.2. Effectiveness of Protection

We ran the Travel Agent application and confirmed that when the Airline web service tried to invoke the **book** service of the Third Party Airline, the request was rejected by SF-Guard on the Airline node. Therefore the client's frequent flyer number could not propagate to the Third Party Airline. Moreover, on each web service node that was on the white list, the business logic could not see the actual value of the sensitive information. These observations show that WS-SensFlow works.

A related question is how to protect SF-Guard itself. Here we assume that there is a TCB on each web service node, and SF-Guard is in the TCB, such that the web service routines can not modify or bypass it. In the current design of SF-guard, we put the entire Web Service Framework into the TCB. However, it is possible that the web service framework itself might be compromised due to bugs in code or due to malicious extensions or malicious configurations. Since we rely on WS-Security processing to protect sensitive information, we have to trust at least a portion of the web service framework. The application of effective techniques to reduce TCB complexity [18] and generate a small and simple TCB is the subject of ongoing research.

5.3. Performance Overhead

We used service completion time as a metric to evaluate overhead introduced by SF-Guard, since it is on the critical path of web service invocation. The service completion time is measured as the elapsed time between a web service routine (e.g., **reserve** service of the Travel Agent) is invoked and the time when the result comes back.

Table 1 shows the completion time of the 8 web service invocations in the Travel Agent example, with and without SF-Guard (Here we assume that the Airline always invokes the **book** service of the Third Party Airline, and the Third Party Airline is not on the black list of the SPEs, so this invocation is allowed). Each invocation is denoted by a requester-provider pair. For example, "Client-T.A." means invocation of the **reserve** service of the Travel Agent by the Client (Due to space constraint, most of the service names have been abbreviated. For example, "T.P.A." represents "Third Party Airline"), and Table 1 tells us that this invocation takes about 793 milliseconds without SF-Guard and 819 milliseconds when SF-Guard is used, therefore the overhead is about 3.3%. Similarly Table 1 shows that after receiving the Client's request, the Travel Agent experiences about 413 milliseconds (without SF-Guard) completion time for invoking the **rent** service of the Car Rental, which in turn invokes the **charge** service of the Credit Card Company and experiences about 305 milliseconds (without SF-Guard) in service time.

From Table 1 we can see that the overhead of SF-Guard on the 8 invocations ranges from 1.6% to 8.3%. These measurements suggest that the overhead of SF-Guard is low, which is not very surprising because SF-Guard mainly performs XML and hash table processing, which is much cheaper than encryption and signing operations by WS-Security. We have not performed much optimization in the implementation (e.g. efficient storing and querying of the SPEs), which may further reduce this overhead.

Table 1: Overhead measurement of SF-Guard (in ms)

	Client –T.A.	T.A.– C.Rtl.	C.Rtl.– Cred.	T.A.– Hotel	Hotel– Cred.	T.A.– Air.	Air.– Cred.	Air.– T.P.A.
Original	793	413	305	123	61	182	60	60
SF-guard	819	422	310	130	63	192	61	65
Overhead	3.3%	2.2%	1.6%	5.7%	3.3%	5.5%	1.7%	8.3%

6. Related Work

Information flow has received considerable attention in computer security research community, with the milestones being Multi-Level Security [8][4], Lattice Model [9], and Java Information Flow [14]. Recent years have seen the application of such models to single-host operating systems such as Asbestos [11]. However, as we mentioned in Section 1, composite web services have unique properties, so the implementation techniques employed by such systems can not be directly applied.

WS-Security [20] is a framework for providing quality of protection to SOAP messages. WS-Trust [23] is an extension to WS-Security that provides means to establish trust relationships among different trust domains. WS-SecureConversation [21] supports the creation and sharing of security context to address the shortcomings of WS-Security. These frameworks or languages as well as P3P [16] and WS-SecurityPolicy [22] can serve as the foundation of implementing WS-SensFlow – for example, they can be used to support SF-Guard. Finally, WS-Trustworthy [27] provides a more generic framework for trusted computing than WS-SensFlow. For example, the information flow constraint specified by a user can be modeled as a specific property in that framework.

There has been significant research on access control in the composite web services [5]. However, they mainly focus on protection of server side resources instead of the sensitive information of a client. The

closest work to ours is a framework proposed by Xu [26] for pulling "models" of composite web services to the client site and checking if they violate the client's privacy policies. This framework assumes that the web service nodes are trusted and the enforcement of privacy policies is above the service nodes. SF-Guard relaxes this assumption and pushes the enforcement into the participating web service nodes. SF-Guard does not focus on the compliance checking, but can leverage on Xu's work for security-policy specification.

7. Conclusion

Current web services enforce data access control on a pair-wise fashion, between service invoker and provider. In dynamically composed services, this kind of access control may expose security-sensitive data (e.g., credit card numbers) to a large amount of untrusted code. This paper presents the SF-Guard architecture to support fine-grain, policy-based access control of security-sensitive data in composite services. SF-Guard is fine-grain because detailed access policy specifications are attached to service invocation messages. These specifications, called WS-SensFlow, are enforced by the participating web service nodes in making access control decisions about the sensitive data.

SF-Guard has been implemented on Axis2 framework (called SG-Wrapper) to support the WS-SensFlow access control policy specifications. The SG-Wrapper applies capability-based encapsulation to enforce the detailed access control. An experimental evaluation of SG-Wrapper using a demonstration Travel Agent composite web service shows strong protection properties and low overhead of a few percent increase in response time.

8. Acknowledgement

This research has been partially funded by National Science Foundation grants CISE/IIS-0242397, ENG/EEC-0335622, CISE/CNS-0646430, AFOSR grant FA9550-06-1-0201, IBM SUR grant, Hewlett-Packard, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. We also thank Qinyi Wu for initial discussion of the problem and the anonymous ICWS'07 reviewers for their insightful comments.

9. References

[1] Amoeba. http://www.cs.vu.nl/pub/amoeba/

[2] Axis2 Architecture.http://ws.apache.org/axis2/1_0/Axis2ArchitectureGuide.html

[3] Apache Axis2/Java. http://ws.apache.org/axis2.

[4] D. E. Bell and L. La Padula. "Secure computer system: Unified exposition and multics interpretation". *T.R. MTR-2997, Rev. 1, MITRE Corp.*, Bedford, MA, March 1976.

[5] Elisa Bertino, J. Crampton, Federica Paci. "Access Control and Authorization Constraints for WS-BPEL". *ICWS 2006*.

[6] M. Chen and J.P. Singh. "Computing and Using Reputations for Internet Ratings". *Proc. 3rd ACM CEC*, 2001.

[7] Trusted Computing Group. https://www.trustedcomputinggroup.org/home

[8] Department of Defense. *Trusted Computer System Evaluation Criteria (Orange Book)*, December 1985. DoD 5200.28-STD.

[9] Dorothy E. Denning. "A lattice model of secure information flow". *CACM*, 19(5):236–243, May 1976.

[10] J. Dorn, P. Hrastnik, A. Rainer. "Web Service Discovery and Composition for Virtual Enterprises". In *Intl. Journal of Web Services Research, Vol. 4, No. 1*, page 23 - 29, 2007.

[11] P. Efstathopoulos, et al. "Labels and event processes in the Asbestos operating system". In *Proc. of the 20th SOSP*, pages 17–30, October 2005.

[12] Butler Lampson. "A note on the confinement problem". *CACM, Vol. 16, Issue 10* (Oct. 1973), 613-615.

[13] Henry M. Levy. "Capability-Based Computer Systems". *Digital Press*, 1984.

[14] Andrew C. Myers, Barbara Liskov. "Protecting Privacy Using the Decentralized Label Model". *ACM TOSEM, Vol. 9, No. 4*, October 2000.

[15] OWL Web Ontology Language Overview. http://www.w3.org/TR/owl-features/

[16] The Platform for Privacy Preferences 1.0 (P3P1.0) Specification, W3C Recommendation, April 2002. http://www.w3.org/TR/P3P/

[17] Axis2/Java – Rampart. http://ws.apache.org/axis2/modules/rampart/1_1/security-module.html

[18] L. Singaravelu, C. Pu, H. Haertig and C. Helmuth. "Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies". In *Eurosys 2006*.

[19] Trail: The Reflection API. http://java.sun.com/docs/books/tutorial/reflect/index.html

[20] Web Services Security v1.1. http://www.oasis-open.org/specs/index.php#wssv1.1

[21] Web Services Security Conversation Language. http://www.w3.org/TR/wscl10/

[22] Web Services Security Policy Language. http://xml.coverpages.org/WS-SecurityPolicyV11-200507.pdf

[23] Web Services Trust Language. ftp://www6.software.ibm.com/software/developer/library/ws-trust.pdf

[24] WebTrust. http://www.webtrust.net/

[25] Li Xiong, Ling Liu. "Peer-Trust: Supporting Reputation-Based Trust for Peer-to-Peer Electronic Communities". *IEEE TKDE, Vol.16, No. 7.* Special issue on Peer to Peer Based Data Management. pp 843-857. July 2004.

[26] Wei Xu, V.N. Venkatakrishnan, R. Sekar, and I.V. Ramakrishnan. "A Framework for Building Privacy-Conscious Composite Web Services". *ICWS 2006.*

[27] Jia Zhang, Liang-Jie Zhang, Jen-Yao Chung. "WS-trustworthy: A Framework for Web Services Centered Trustworthy Computing". In *SCC 2004.*