CERIAS Tech Report 2006-18

INFORMATION LEAKS AND SAFE WEB SERVICES

by Ashish Kundu

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Information Leaks and Safe Web Services

Ashish Kundu
Department of Computer Science, Purdue University
ashishk@cs.purdue.edu

Abstract

The paper shows that information leaks are inherent in object models based on subtyping and inclusion polymorphism. Web services interact with other systems across organizational boundaries using such an object model. In the context of web services, information leaks pose serious security and privacy concerns. A safe web service is one which neither is a source of any information leak nor exploits any information leak. The paper defines properties of such a safety model and proposes mechanisms to enforce the safety requirements. Leaks inherent in the programming paradigm however cannot always be completely masked while keeping the desired interoperability and flexibility of services intact, especially in compositional scenarios. Therefore the paper also proposes use of processes of service certification and versioning aided by data flow analysis as measures against, and a cost estimation model in case of information leaks.

1. Introduction

Web services [15] are fast becoming the standard model for development and deployment of distributed applications [6]. With a growing need to provide value-added services, cross-organizational service delivery is gaining momentum. Web services use SOAP [16] for interaction among various parties (web services and clients) deployed across organizational boundaries (Figure-1). The parties interact among each other for various activities such as discovery, registration, authentication, request and delivery of services.

In the web setting, these distributed applications involve interactions between various untrusted parties (deployed on cross-organizational domain). Interactions between these parties - client and service, service and service - involve flow of control and data (information) among them. Cross-organizational flow of information raises privacy and security concerns.

Object model is the underlying model for data (information) flow among various parties – services and clients. Objects are exchanged as parameters and results of invocations. The flow of objects from one party to another is made feasible through support of subtyping and inclusion polymorphism [1]. Subtyping facilitates compatibility between interfaces. Even if the parameter types in the interfaces of two interacting parties are not identical, the information flow is still possible without any explicit type transformation. If the type of the transmitting party is a subtype of the receiving type, then information can flow from the sender through the receiver interface. Inclusion polymorphism supports implicit type conversion from a subtype to its supertype.

Web services and clients are commonly developed (synthesized) using languages (*e.g.,* Java, C++) that support inclusion polymorphism and subtyping. In this paper we show that interactions based on the object-model using subtyping and inclusion polymorphism is not *safe* in terms of security and privacy. Default type conversion of an object of a subtype to one of its supertypes leads to information leaks. Despite the type conversion, the object holds all the member fields including embedded objects and methods belonging to subtype. This extraneous information with respect to the supertype gets transmitted to the party, which might be an untrusted destination.

In an untrusted environment, service providers (organizations) might carry out unauthorized operations on units of information (objects) received from other parties (services or clients). Untrusted environments comprise of untrusted clients, web services or manipulated execution environment; the underlying middleware and execution environment such as the marshaling library and Java Virtual Machine can be manipulated to carry out unauthorized operations.

Unauthorized operations are the operations on information that are not necessary to be applied on an object received from another party for the purpose of providing the desired service. Examples of unauthorized operations include extraction, copying, cloning or modification of extraneous data in an object partially or fully. An untrusted party is one that carries out unauthorized operations by design or inadvertently on an object received from another party.

The members of a subtype that do not belong to its immediate supertype are called specialized members of that type. The specialized members of an instance ($objType_1$ of type $SubSubObjType_1$; see Figure-2) are also referred to as extraneous members in the context of an interaction between two parties. For example all specialized members of an instance of type $SubSubObjType_1$ with respect to its supertype $ObjType_1$

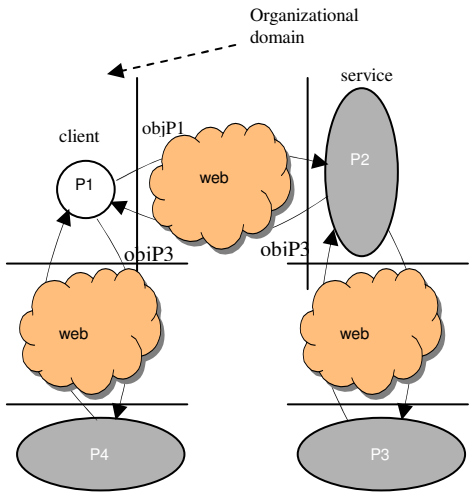

Figure 1. Interactions between various parties.

are extraneous members for the interaction between P_1 and P_2 (Figure-1). By members we denote all the variables, member methods in an object or a type.

Automatic and dynamic service discovery and service request mechanisms employ type matching that exploits the supertype and subtype relationship. Automation of the process of aggregation (or composition[1]) of web services has gained importance with the need for value-added higher order services [12]. With the advent of adhoc networks, pervasive and mobile computing models and dynamic integration of services, there is need to support dynamic interactions among the services and the clients need to be supported, but in a secure, privacy-preserving manner. In order to facilitate a safe computing environment in these contexts, the paper attempts to provide mechanisms for effectively removing information leaks at the level of programming model and life cycle of web services.

Export of only those members that belong to the required supertype *S* through cloning is not a viable solution in order to prevent leak of extraneous members. This is because, 1) the supertype(s) should provide cloning mechanisms, 2) the cloning mechanism of *S* ($ObjType_1$ in Figure-2) should be directly accessible to an object of a type ($SubSubObjType_1$) that may not be a direct descendant of *S* in the type hierarchy and 3) the process of cloning especially deep cloning is expensive. (1) is not pragmatic because of legacy services, and existing types may not support cloning. For similar reasons marshalling and serialization mechanisms cannot selectively export only the required members.

The paper shows that information leaks are inherent in the object model and are sources of security and privacy breaches. The other contributions of the paper are

- a safety model for web services against leaks,
- methodologies to prevent information leaks, through member masking and partial encryption of objects.
- application of certification and versioning in evaluation of the degree of safety in web services and
- approaches for estimation of cost associated with information leaks.

1.1 Interaction Model

Let P be a set of interacting parties. Nothing is assumed about the interaction model and its synchronous properties. Let $P_i \in P$ be a party. Let M_{ij} represent the interface exported by web service P_j, which P_i invokes to interact with P_j. M_{ij} is $((T_{1j}, T_{2j}, \ldots, T_{mij}), R_{ij})$, where R_{ij} be the return object type for interface M_{ij}. An interaction between P_i and P_j, I_{ij} is $(((T^i_{1j}, O^i_{1j}), (T^i_{2j}, O^i_{1j}), \ldots, (T^i_{mij}, O^i_{mij})), (R^j_{ij}, S^j_{ij}))$. O^i_{kj} is the object sent to P_j by P_i in place of T_{kj} in M_{ij}, or *null* if the object is *null*. T^i_{kj} is the type from which O^i_{kj} is instantiated from. R^j_{ij} is the return type of the object S^j_{ij} sent by P_j to P_i as part of this interaction I_{ij}. T^i_{ij} *and* T_{ij} are identical types, if and only if O^i_{ij} is instantiated from type T_{ij}; *i.e.*, O^i_{ij} contains no object or data that is extraneous with respect to type T_{ij}.

1.2 Outline of the Paper

The outline of the paper is as follows. Section 2 illustrates the various kinds of information leaks, their originations and exploitations. Section 3 introduces the notion of information flow in the context of web services. Safe web services computing model is proposed in the next section. Properties that govern safe information flow and mechanisms to enforce them in web services paradigm are described in Section 4.1 and 4.2. Section 4.2 proposes the use of certification and versioning of web services with the aid of static program analysis in order to evaluate the degree of safety of a given web service. A cost estimation model for information leaks is proposed in 4.3. Section 5 discusses the concepts and techniques introduced in the paper. Related prior work is in the next section and Section 7 concludes the paper.

2. Information Leaks

Consider an interaction between parties P_1, P_2, P_3 and P_4 as shown in Figure-1. Table-1 lists the method signatures of these parties. P_1 implements a callback method ($callback_1$) for asynchronous responses from

[1]Strictly, aggregation and composition refer to different semantics. However in the web services context (and so in this paper) these two terms have been used pretty much interchangeably.

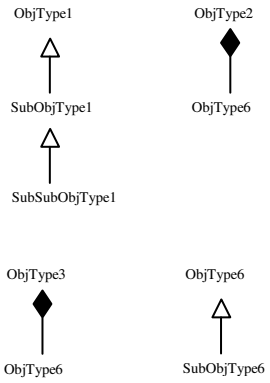

Figure 2. Type Hierarchies.

services. P_2, P_3 and P_4 implement methods $method_2$, $method_3$ and $method_4$ respectively, for service requests (mechanisms for interaction).

Table 1. Method Signatures of Interacting Parties

Party	Return Type	Method	Parameter Type(s)
P_1	*void*	$callback_1$	$ObjType_3$
P_2	$ObjType_3$	$method_2$	$ObjType_1$
P_3	$ObjType_5$	$method_3$	$ObjType_2$, $SubObjType_1$
P_4	$ObjType_5$	$method_4$	$ObjType_3$

An interaction is said to be of compositional semantics (or just compositional) if it involves more than one service. In Figure-1, interaction between P_1 and $\{P_2, P_3\}$ - an aggregate web service - is a compositional interaction. An interaction is said to be of elemental semantics (or just elemental) if it involves one service and one client as the two interacting parties. In Figure-1, interaction between P_1 and P_4 is elemental.

Compositional: P_2 and P_3 comprise a composite web service. An interaction between P_1 and this service is called compositional. P_1 sends $objP_1$ to P_2; P_2 sends $objP_1$ and $objP_2$ to P_3. Instance $objP_2$ has $objP_6$ of type $SubObjType_6$ embedded in it. P_3 creates $objP_3$ of type $ObjType_3$. It extracts the object of type $ObjType_6$ embedded in $objP_2$. This is actually the instance $objP_6$ of type $SubObjType_6$. P_3 embeds $objP_6$ in $objP_3$. P_3 returns $objP_3$ to P_2, which in turn returns it to P_1.

In this stage, information leak occurs at three places. The message from P_1 to P_2 leads to one of these. P_1 sent a specialized object $objP_1$ of type $SubObjType_1$ to P_2, while P_2 was expecting an instance of type $ObjType_1$. However since P_1 and P_2 are assumed to mutually trusted parties, this information leak is *locally* harmless. This locally harmless leak would have led to a serious leak (*transitive* leak), if P_3 would have expected an instance of type $ObjType_1$ instead of the type $SubObjType_1$. The second one occurs when P_2 sends $objP_6$ embedded in $objP_2$ to P_3. P_3 is an untrusted party for P_2. Therefore this leak is a serious one, especially since the extraneous information in $objP_6$ is sensitive information such as social security number of a student (see Figure - 3). The third leak occurs between P_2 and P_1. P_2 returns $objP_3$ that embeds $objP_6$ of type $SubObjType_6$, while P_1 expects an instance of type $ObjType_6$ embedded in $objP_3$. Again P_1 and P_2 are mutually trusted parties, therefore it is locally harmless. However this harmless leak would lead to a serious information leak as shown next.

Elemental: The party P_4 is an elemental service (not a composite service). The interaction between P_1 and P_4 is called elemental interaction. In this second stage of the interaction, P_1 interacts with P_4. P_1 passes $objP_3$ received during the first stage of interaction to P_4. P_4 is not a trusted party for P_1 and P_4 expects an instance of type $ObjType_6$ embedded in an object of type $ObjType_3$. In addition, P_1 is not the creation point for $objP_3$. Therefore even if P_1 were capable (it had the intelligence built into the code) of preventing the information leak, it was not able to prevent this leak. The leak could have been prevented if in the first case P_1 would have only received an instance of type $ObjType_6$ instead of type $SubObjType_6$ in $objP_3$. The leak could have been prevented if P_1 would have carried out type matching and inference.

The information leak from P_1 to P_2 occurred due to polymorphism directly, while the other leaks occurred due to polymorphism but indirectly (added by object composition). We can now formulate the notion of information leaks in the context of the web-services interaction model (Section 1.1).

2.1 Formal Definition of Information Leaks

Let $T^i_{ij} - T_{ij}$ denote all the extraneous members (top-level as well as transitive) of T^i_{ij} that is not expected as part of the definition of type T_{ij}.

Leak $L_{ij} = ((L^i_{1j}, L^i_{2j}, \ldots, L^i_{mij}), L^i_{Rij})$, where L^i_{ij} is defined as follows

- $L^i_{ij} = T^i_{ij} - T_{ij}$.
- $L^i_{ij} = \{f^{ik}_{ij} \mid f^{ik}_{ij}$ is a set of k'th members - that exist in T^i_{ij} but not in $T_{ij}\}$. f^{ik}_{ij} might consist of only a top-level member of T^i_{ij} or might be a set of all members that belong to an embedded type E^{ik}_{ij} in T^i_{ij} such that the corresponding embedded type E^k_{ij} in T_{ij} is a supertype of E^{ik}_{ij}. For example, in the interaction from P_1 to P_4 in Figure-1, an instance of type $SubObjType_6$ (E^{ik}_{ij}) is transmitted against the expected type of $ObjType_6$ (E^k_{ij}) embedded inside $objP_3$.
- L^i_{ij} = *null* (or *empty*) if T^i_{ij} is *null* or T^i_{ij} *and* T_{ij} are identical object types,

- L^i_{ij} = *effectively-null* if the values for the members in T^i_{ij} other than those expected by T_{ij} are *in-accessible* by the receiver.
- L^i_{ij} = *non-null*, if T^i_{ij} *contains* more information that does not exist in T_{ij}.

L_{ij} is said to be *null* if each of its elements is also *null*.

2.2 Exploitation of Information Leaks

Malicious parties involved in interactions that lead to information leaks, and their hosting environments would try to exploit these leaks in the objects they receive. Hosting (execution) environment of a web service includes the dependent libraries, SOAP server, XML processing middleware such as marshalling libraries and virtual machine (such as Java Virtual Machine). Each of these modules independently is capable of exploiting leaks in messages.

2.2.1. Malicious Parties. Malicious web services might request more information from the interacting party. This can be achieved by declaring the type of a formal parameter in their interface(s) to be a subtype of the actual type necessary to provide the service. This is called as type deception. Malicious parties (including clients) could also peek into the objects they receive as part of messaging with the other parties for any potential information leak even when type deception is not employed. Some of the possible ways in which a malicious party could detect presence of leaks are type inference, and size matching. Type inference would help the party in determining if the type of the object is a subtype of its expected type. A party with a good knowledge of what could be an average size of the object could compare the size of received object and infer with some probability if there is an information leak. Once the presence of an information leak is detected, the party would then proceed with other techniques such as reflection and cryptanalytic attacks on encrypted objects, to operate on the extraneous data.

2.2.2. Malicious Organizations. Service hosting modules can be engineered to detect and exploit existing information leaks through various methods. For example, Java Virtual Machine used to host Java based web services and clients can be engineered to operate on all the members of objects received irrespective of whether they are public or non-public. The modified JVM is then capable of operating on the private and protected members of an object unlike in the case of malicious parties. Since the underlying hosting environment and virtual machines are engineered, almost any operation on extraneous members of the objects could be carried out,

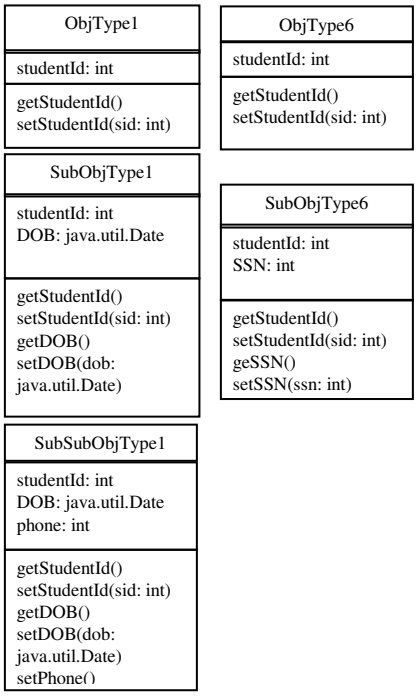

Figure 3. Class Definitions.

including cloning of objects even if cloning is not supported.

3. Safe Information Flow

An interaction is safe if does not lead to any information leak (neither local nor transitive). Interaction I_{ij} is safe if and only if each element of leak L_{ij} is *null*. If some information received by P_i from another party P_k is sent to another party P_j in an interaction I_{ij} is in leak L_{ij} then such a leak is an indirect (transitive) leak; otherwise the leak is a direct (local) leak. A local interaction I_{ij} is one in which P_i sends data to P_j such that none of the data has been created at P_k ($i \neq k$). All other kinds of interactions are transitive in nature.

Information flow among a set of parties is said to be safe if and only if

- all the parties are mutually trustable, i.e. they do not carry out any unauthorized operations on the messaging objects or
- any interaction between the parties is safe.

However it can not always be guaranteed that all the interacting parties are mutually trustable. In order to prevent information leak, there is a need to support mechanisms to avoid extraneous information from being transferred altogether or being operated upon. It is also desirable to minimize the cost of information leak to another party during an interaction, if information leak

cannot be prevented or avoided. Section 4.3 presents a cost estimation model for information leaks in web services context.

4. Safe Web Services

Information leaks at both elemental and compositional interactions should be prevented. Safe parties are therefore essential for the development and deployment of the desired service architecture. A party is said to be safe if and only if it satisfies the following safety properties

- *Leak-Source Property* (*s-property*): the party is not a source for any information leak (as defined below) and
- *Leak-Exploitation Property* (*e-property*): the party and its execution environment do not exploit any information leak.

A party P_i is not a source of any leak, if each of its interaction I_{ij} with any party P_j is safe in terms of information flow, *i.e.* leak of I_{ij}, L_{ij} is *null*. A party does not exploit an information leak existing in an interaction in any manner if the party itself or the underlying execution environment does not carry out any unauthorized operation on the extraneous information.

In the following sections we would explore various approaches for safe information flow among various parties.

4.1 Leak-Source Property

Leak-Source property is said to be enforced if and only if both local and transitive leaks do not manifest in any interaction. These two factors lead to two corresponding sub-properties: local leak source property (*ls-property*) and transitive leak source property (*ts-property*).

4.1.1. Local Leak-Source Property. If a party ensures that for every interaction I_{ij} it will have with any other party, the objects it sends have identical types as expected by the receiving parties then it is said to satisfy the *ls-property*. If each of the objects of type T^i_{kj} ($1 \le k \le m_{ij}$) does not contain any data that is a result of any other interaction in which P_i has been involved in, then P_i can be implemented such that it only synthesizes objects of type $T_{kj} \in M_{ij}$.

However P_i might re-use some object O_{ih} partially or completely that has been part of a local interaction I_{ih} (P_k is not necessarily different from P_j) in synthesizing an object O_{ij} for another local interaction I_{ij}. Say D_{ih} is the information such that it belongs to both O_{ih} and O_{ij} (D_{ih} *subset of* O_{ih}, O_{ij}). Therefore a party Pi must be implemented such that the objects it synthesizes for any

```
for each specialized member variable X do {
    if there is a setter function // Java Bean
        invoke method setX(random or undefined value);
    else mask its value by X := random or undefined value;
}
```

Figure 4. Sketch for mask method

local interaction I_{ij} in a conservative manner. By conservative, it is meant that P_i extracts only the required data D_{ih} from O_{ih}.

Therefore the party – the web service or the client – must include selective data extraction from objects sent as part of local interactions. Such a party always satisfies *ls-property*. We call such parties as *locally-safe*. Local interactions of locally-safe parties are called safe-local interactions.

4.1.2. Transitive Leak-Source Property. Enforcing *ts-property* in interactions between web services and its clients is more difficult than enforcement of *ls-property*. O^h_{ih} is an object received from another party P_h as part of interaction I_{ih} initiated by P_i. Let the type of O^h_{ih} be T_{ih}. O^i_{kj} is an instance of type T^i_{kj} expected by P_j as part of interaction I_{ij} from P_i. D_{ih} is a set of information such that D_{ih} *subset of* O^h_{ih}, O^i_{kj}. D_{ih} consists of only

1. either some publicly accessible data in O^h_{ih} or
2. is an instance of T_{ih} or
3. is some supertype S_{ih} of T_{ih}.

(1) P_i extracts the publicly accessible data required and adds them to O^i_{kj}. (2) & (3) P_i embeds O^h_{ih} or its clone. In case of (3), O^h_{ih} is cast to the type of S_{ih} (the type expected by P_j as part of I_{ij}).

The *ts-property* is violated when the embedded object (O^h_{ih}) is of a subtype (T_{ih}) of the type expected by the receiver (S_{ih}). In order to satisfy the *ts-property*, leak L^h_{ih} (= T_{ih}–S_{ih}) is to be *effectively-null*. L^h_{ih} cannot be *null* because T_{ih} is not identical to S_{ih}. However P_i does not have access to the non-public members of O^h_{ih}. Therefore in order to make L^h_{ih} *effectively-null*, mechanisms should be introduced to make the values of the non-public specialized members of T_{ih} inaccessible. We propose two methods for making the values inaccessible: masking and encryption of the members.

Member masking: Each object-type defined to be used in the context of web-services interactions, should provide a mechanism for masking all the specialized members in this type. In Java, every class *C* is extended from *java.lang.Object*; thus any member in *C* is a specialized member. In C++, there is no root class; thus if *C* is inherited from *C'*, then every member in *C* that is not in *C'* is a specialized member.

```
for each specialized member variable X do {
    encrypt value of X by
    X := encrypt value of X using key K;
}
```

Figure 5. Sketch for encrypt method

T_{ih} is a subtype of S_{ih} (defined above). Each object-type (class) T_{ih} and S_{ih} defined to be used for the interaction between web-services and clients, should define a method with the following pattern: **mask***typename*(), *typename* being T_{ih} and S_{ih}, respectively for definition of object-types T_{ih} and S_{ih}. *mask*T_{ih}*()* masks all the specialized member variables with some random or boundary values of T_{ih}. *mask*T_{ih}*()* can be sketched as in Figure-4.

Partial encryption: Specialized members can be made in-accessible by encrypting them. Each defined type implements a method **encrypt***typename(), typename* being T_{ih} and S_{ih}, respectively for definition of object-types T_{ih} and S_{ih}. *encrypt*T_{ih}*()* encrypts all the specialized member variables of T_{ih} using some key. *encrypt*T_{ih}*()* can be sketched as in Figure-5. Key can be defined to be random key, or can be determined using some key management technique. The latter is useful especially in compositional interactions. Object O^h_{ih} sent to P_j as an instance of type S_{ih} may be sent to P_k, which might use it as an instance of type T_{ih}.

A client for a web service is synthesized based on the service definition (WSDL [17]). Programming methodologies for the web services and their clients should

- satisfy the *ls-property* by imposing a tight type matching.
- implement either of *mask* or *encrypt* method for each object-type they define in order to facilitate conformation to *ts-property*.

Tight type matching requires only those objects to be sent from client to the web service, which are instances of identical types as they are expected by the web service interface.

Building of compositional (or composite) web services should also follow the above procedures. Since there is less independence in defining the object-types in this case, type transformation must be carried out explicitly so that information does not get leaked. This requirement however limits the scope of completely automating the services composition. Decentralized orchestration of composite services [11] should be avoided if the various properties governing information leaks could not be satisfied altogether. In case of dynamic composition of services or dynamic interactions that were not anticipated during development of the service or their deployment, it is desirable to estimate the cost of any information leak that could not be prevented. Section 4.4 discusses a general estimation model for this purpose.

4.2 Leak-Exploitation Property

The *e-property* is said to be enforced if the complete execution environment is honest; *i.e.* it would not exploit any potential information leaks. In order to satisfy this property, a given party and its execution environment must either be a trusted with respect to the other interacting party. Web services should only carry out interactions with a client that are safe with respect to information flow. A web service P_j must always send data objects to a client P_i as part of an interaction I_{ij} that is a subset of the return type R^j_{ij}. This would ensure leak L^i_{Rij} to be *null*. In this case *e-property* need not be enforced on the parties that are clients.

However web services need to be developed, deployed and maintained such that *e-property* is satisfied at each stage. Section 4.3 discusses approaches to evaluate degree to which the *e-property* is satisfied by a web service and its execution model.

It is undecidable [19] to automatically analyze and prove that a given web service and its host software satisfies *e-property* or in other words, is safe. Therefore various processes during services development, deployment and maintenance phases could be applied for this purpose. Estimation of safety should be the responsibility of each service provider to employ human experts and optionally, tools based on program analysis. Certification of web services, underlying middleware and execution environment is an important approach. Versioning of web services, use of version information during service invocations in order to prevent unsafe information flow is another important approach. Static program analysis of web services and underlying dependencies is very useful for both certification and versioning processes.

Estimating and guaranteeing the degree of safety of a web service with respect to the *e-property* is achieved by three ways – certification, versioning and cost estimation of information leaks.

4.2.1. Certification. Certification of web services can be used to determine how safe is a web services that would not exploit information leaks. It also provides some guarantee on the outgoing interactions – this service would not send extraneous information out to another party. It is considered safe to interact with a certified web service even with no prevention mechanisms for information being used. One or more accredited cross-organizational authorities can verify the safety claims by service providers independently and issue certificates. These authorized institutions should also review

modifications to a web service periodically and re-evaluate the certificate.

The process of certification verifies if the program carries out any unauthorized or malicious operations on the data that it receives. Static analysis (Section 4.2.3) could be used for this purpose.

Certificates for safety can be organized in hierarchical manner. A higher level certificate guarantees more safety and subsumes the degree of safety guaranteed by a lower level certificate. Root of certificate hierarchy defines highest-level of safety.

4.2.2. Versioning. Maintenance and evolution of web services would be more frequent than the frequency of issuance of certificates for trust-able behavior. Every change to the web service implementation and deployment configuration that might possibly affect the *e-property (and s-property*) has to be given a new version.

During setup of a session or invocation by a party, it should make sure that it is using the appropriate web service. Appropriateness of a web service is based on whether the client's interactions with that service would be safe. Versioning and certification both used together would help authenticate the appropriate web service with the client. This technique can be symmetrically used for the other party, if it is a web service. A proper versioning mechanism would prevent the service provider or developer from replacing original web service by a malicious one or an untested, uncertified service.

4.2.3. Static Program Analysis. Static program analysis [9][14] is an aid but not a foolproof method for detecting of information leaks and violations of *s-property* and *e-property*.

Liveness analysis [14] of parameters (and their internal members recursively) of a service method could determine if a member is extraneous or not. If a member variable is not live at any point in the program, then it is a candidate for being labeled as an extraneous member. However this procedure could be fooled by adding statements that uses each member variable at the end of the program.

Program slicing [13][14] can be used to determine if all the member variables affect the output or state of the program. Given the set of variables that hold the result and effects of a request processing, backward slicing on those variables could determine which member variables affect them directly or indirectly. One such variable is the return variable. However effectiveness of slicing is limited by the presence of aliases.

Operations such as cloning of parameter objects, their manipulations should be detected through program analysis and reported back to the human experts for semantic analysis of such code. Code segments carrying out type conversions on parameter objects and their aggregation into other objects should also be detected. Aggregate (or composite) objects containing references to parameter objects (especially candidate extraneous members) should be identified and guarded for detection of their cloning, storage or transmission.

4.3 Cost of Information Leaks (KLIC)

Cost of information leaks (KLIC) can be used to determine whether one party should interact with another party. It can be used in aggregation of web services or in dynamic binding to web services. A cost estimation model is proposed in this section for quantitative evaluation of the negative implications of information leaks. The models would differ based on the business context they would be used, the sensitivity of the messages among other parameters.

Naive model: One of the various possible naïve measures of extent of information leak could be based on the size of extraneous information being transferred in interaction I_{ij}. Using *size* that denotes the actual size of the object, the cost of a leak for I_{ij} is

$$KLIC_{ij}=(\Box_{1\Box k\Box mij} size(O^{i}_{kj})-\Box_{1\Box k\Box mij} size(O_{kj}))/\Box_{1\Box k\Box mij} size(O_{kj}) \quad ...(I)$$

A similar measure can be defined for leak in return object O^{i}_{ij}. Other possible factors could be the number of member fields being part of a leak.

Weighted model: Sensitivity of each information unit getting leaked differs. The implication of leak of one information unit is proportional to its degree of sensitivity. The weighted model takes "weight" as a quantity proportional to the degree of sensitivity. For an interaction I_{ij}, the cost of a leak is

$$KLIC_{ij}= \Box_{1\Box k\Box mij} \Box_{1\Box l\Box nk} (D^{l}_{k} + V^{l}_{k} + M^{l}_{k})*W^{l}_{k}. \quad ...(II)$$

W^{l}_{k} is a quantity proportional to the degree of sensitivity of information leaked in conjunction with the level of trust among the two parties. Level of trust among the two parties may be determined by use of some model of trust estimation. W^{l}_{k} might be dependent on temporal and contextual factors as well. D^{l}_{k} is the penalty for releasing the meta-data about a member field f^{il}_{kj}. The meta-data for a member includes the field-name, its type and size. M^{l}_{k} is the cost of releasing information about the method signature and definition if f^{il}_{kj} is a method. If f^{il}_{kj} is a method, then its cost would depend on whether it is public or private; for private methods the cost would be higher. V^{l}_{k} is the cost of releasing the value (or method definition) of f^{il}_{kj}. If the member f^{il}_{kj} is a non-primitive data type (*e.g.*, an embedded object), then V^{l}_{k} depends on the cost of all internal members of f^{il}_{kj} and is recursively computed (possibly in a bottom-up manner).

5. Discussion

The occurrences of information leaks can be controlled by following programming methodologies that tend to satisfy *ls-property*, *ts-property* and *e-property*.

Member masking and partial encryption are good for protecting the associated values of extraneous members. Objects on which partial encryption is applied are vulnerable to cryptanalytic attacks to reveal the leaks. The receiving party can carry out various offline attacks and dictionary attacks. Cryptanalytic attacks would be more focused on the encrypted segments. This would make these attacks more effective (at least theoretically). The member masking technique exposes the meta-data of extraneous members – type and name of the members. Albeit they are some information that is leaked, such leaks are rarely any major reasons for security and privacy infringements.

Techniques for web services composition need to bring the factor of information leaks into their composition and optimization framework. The KLIC model can be used as it is or with appropriate modifications, in such an optimization framework that tries to minimize or keep the cost of information leak within some bounds. The logical and physical composition stages [12] have to attempt for tight type matching. The physical composition stage has to enforce *ts-property*. Services to be deployed in enterprise (thus trusted) domains need not be optimized towards information leaks. However as argued earlier, information leaks have serious implications in cross-organizational service offerings.

A version could be a key that is generated through hashing of the whole source code of the program. Certification of a service should be based on how safe is the service against information leaks. Type deception – asking for a subtype, while the party only needs less information as supported by a supertype is to be detected during process of certification and program analysis.

6. Related Work

To the best of the author's knowledge, this is the first work that shows that object-based messaging among distributed components using inclusion polymorphism leads to potential information leaks. Use of web services versions in authentication between two parties for future interactions seems to be a new contribution from this paper.

Security and privacy in web services model has received wide attention in the research community recently ([7][8]). It had been assumed that the underlying object-oriented paradigm does not lead to information leaks especially in a distributed setting. The messaging paradigms have relied upon the underlying object model.

Secure information flow in the context of type systems has been studied extensively. [2] proposes a type system for this purpose. JFlow is a language as an extension to Java [5] in order to support flow of information among in a controlled and secure manner. However the problem of information leaks (in-secure information flow) arising out of inclusion polymorphism [1] has not been studied in literature.

Use of static analysis for certification of the security of programs has been studied in literature [4][5]. Use of static analysis for security evaluation of programs is also recommended in [4][5] and JFlow has been created to facilitate the same. These techniques can be extended to certification of web services. However the issue of information leaks has to be accounted for during certification. [3] proposes analysis of flow of information in order to detect attacks against a program. The analysis technique can be used in the context of leaks.

7. Conclusion and Future Work

The paper shows that information leaks through object-transfer are inherent in object models based on subtyping and inclusion polymorphism. Within the premises of the popular object models, we have proposed some approaches for effective removal of information leaks. Leaks inherent in the programming paradigm however cannot always be completely avoided while keeping the interoperability and flexibility of services, especially in compositional scenarios. Therefore we have proposed processes of service certification, versioning as measures against, and a cost estimation model in case of information leaks.

The proposed model for safe information flow and proposed development guidelines should be incorporated in web services development toolkits such as IBM ETTK [18], for development of safe web services.

Key management in partial encryption has implications on data flow in compositional interactions. Certification and versioning of web services need to be studied further and should be employed in web services life cycle management. The version and certification based mutual authentication of parties before or during interactions can be extended to services model in general, where information leak is a potential problem.

8. References

[1] L. Cardelli and Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", *Computing Surveys*, ACM , Vol. 17, No. 4, December 1985, pp. 471-522.

[2] K. Honda and N. Yoshida, "A uniform type structure for secure information flow", *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Portland, 2002, pp. 81-92.

[3] R. Yokomori *et al*, "Analysis and Implementation Method of Program to Detect Inappropriate Information Leak", *Proceedings of the Second Asia-Pacific Conference on Quality Software*, 2001, pp. 5.

[4] A. C. Myers and B. Liskov, "A decentralized model for information flow control", *Proceedings of the sixteenth ACM symposium on Operating systems principles*, 1997, pp129-142.

[5] A. C. Myers, "JFlow: practical mostly-static information flow control", *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, San Antonio, 1999, pp 228-241.

[6] F. Leymann, "Web Services: Distributed Applications Without Limits", *BTW Conference, 2003*.

[7] C. Li and C. Pahl, "Security in the Web Services Framework", *Proceedings of the 1st international symposium on Information and communication technologies*, Dublin, 2003, pp 481-486.

[8] K. Bhargavan *et al*, "Verifying policy-based security for web services", *Proceedings of the 11th ACM conference on Computer and communications security*, Washington DC, 2004, pp. 268-277.

[9] W. Masri and A. Podgurski, "Using dynamic information flow analysis to detect attacks against applications", *Proceedings of the 2005 workshop on Software engineering for secure systems-building trustworthy applications*, St. Louis, 2005, pp. 1-7.

[10] G. Smith and D. Volpano, "Secure information flow in a multi-threaded imperative language", *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, San Diego, 1998, pp. 355-364.

[11] M. G. Nanda *et al*, "Decentralizing execution of composite web services", *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Vancouver, 2004, pp. 170-187.

[12] V. Agarwal *et al*, "A service creation environment based on end to end composition of Web services", *Proceedings of the 14th international conference on World Wide Web*, Chiba, 2005, pp. 128-137.

[13] Frank Tip, "A survey of program slicing techniques", *Journal of Programming Languages*, Vol. 3, No. 3, 1995, pp. 121-189.

[14] Muchnik, S., *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.

[15] World Wide Web Consortium, Web Services Architecture, http://www.w3.org/TR/ws-arch/.

[16] World Wide Web Consortium, Simple Object Access Protocol, http://www.w3.org/TR/2000/NOTE-SOAP-20000508.

[17] World Wide Web Consortium, Web Services Description Language, http://www.w3.org/TR/2006/CR-wsdl20-primer-20060106/.

[18] IBM ETTK Toolkit, http://www.alphaworks.ibm.com/tech/ettkws.

[19] G. Ramalingam, "The undecidability of aliasing", *ACM Trans. Program. Lang. Syst.*, Vol. 16, No. 5, 1994, pp. 1467-1471.