Trust, But Verify: An Approach for Realizing Privacy-Preserving Web-Based Services*

Wei Xu
Department of Computer Science
Stony Brook University
Stony Brook, NY 11790-4400
weixu@cs.sunysb.edu

V.N. Venkatakrishnan
Department of Computer Science
University of Illinois at Chicago
Chicago, IL 60607
venkat@cs.uic.edu

R. Sekar
Department of Computer Science
Stony Brook University
Stony Brook, NY 11790-4400
sekar@cs.sunysb.edu

I.V. Ramakrishnan
Department of Computer Science
Stony Brook University
Stony Brook, NY 11790-4400
ram@cs.sunysb.edu

ABSTRACT

The rapid growth of web applications has raised significant concerns regarding personal information privacy. Currently, such concerns have addressed by non-technical mechanisms that involve privacy policy statements that were stated in natural language. Such approaches have remained ineffective as they have involved users to understand the complex phrasing of privacy policies which are verbose, difficult to understand and sometimes inconsistent. In this paper, we present a framework for empowering users with mechanisms to control the privacy of their personal information when using it in web-based services. The novelty of our framework is built upon a judicial combination of *trust* and *verification*, and provides a technological basis for consumers and service providers to verify compatibilities between privacy policies and terms of usage. We describe our framework and report our experiences of an implementation in the context of Personal Information Assistants, a mechanism for building composite web services from primitive services.

1. INTRODUCTION

The past decade has witnessed a phenomenal growth in the number of users who routinely use the web to obtain information, conduct research, or carry out financial transactions. While the ability of the web to provide customized information and financial services has boosted personal and business productivity, it has raised significant concerns regarding an individual's privacy. The privacy concerns are exacerbated further with the advent of more complex services (such as *Orbitz.com*) that involve multiple service providers, as it raises the potential of personal information being shared across these providers in ways that weren't intended by the owner of the information.

To illustrate the increased privacy concerns raised by composite web-based services, consider a web-based service that allows an individual to research and order over-the-counter medication (or health supplements) at a low cost, while ensuring that there are no adverse interactions based on the individuals current prescription. Such a service will require access to the individual's personal medical history, and will also require interaction with other entities such as web-based medical references, discount pharmacies and credit-card companies. Users may not want their personal health history to be shared with discount pharmacies or credit card companies. Similarly, users may not want their credit card information to be shared with providers of free on-line information.

To address consumer concerns on information privacy, web-based service providers have taken the steps to conspicuously identify and display their policies regarding the storage and use of personal user information. Unfortunately, these steps have not translated into increased user confidence in the privacy of their personal information. This is due to a combination of two factors. The first concerns the verbose and imprecise nature of privacy policies. Usually these policies are stated in natural language, and challenge a user with issues such as imprecise and ambiguous phrasing and legal terminology. Typically these policies resemble the dreaded software End-User License Agreements (EULAs). Users simply do not have the time or know-how to understand or interpret the varying privacy policies across different web sites. The second problem concerns the lack of technological means to monitor conformance with stated policies. This factor tends to remove any significant liability on the part of service providers for all but intentional, large-scale breach of their stated policies. As a result of these two factors, we are witnessing a continued rapid erosion of user confidence on the privacy of their personal information.

*This research is supported by a grant from Computer Associates and an ONR grant N000140110967.

The potential of advanced web services will remain unrealized unless user privacy concerns are addressed satisfactorily. In this paper, we put forth our approach which is concerned with the development of a *practical framework* in which:

- consumers can express their *privacy policies* in a formal way,
- service providers can express their terms of usage using a *model,*
- both providers and consumers have a technological basis to verify compatibility between policies and terms of usage.

The technology behind the framework automates compatibility checking between policies and models. If there is an incompatibility, the consumer is informed how she can refine her policies in order to use the service. If she does not want to change her policies in any way, the approach passes on additional privacy requirements to the provider. Service access can continue in case of incompatibilities only if the consumer relaxes her policies, or the provider honors additional consumer privacy requirements.

The key idea behind our approach is a judicial combination of *trust* (on service providers to accurately specify use of consumer data) and *verification* (for compatibility resolution). This combination enables our approach to support privacy preservation without requiring access to proprietary code that implement the service. This is in contrast with approaches such as static analysis for information leaks, which have relied on availability of source code that implements the service. (Based on current trends, one can safely say that disclosure the source code and the algorithms involved of any business service is a rare possibility.) Other solutions such as P3P [2] (Platform for Privacy Preferences) have aimed at web resources such as cookies and files. P3P has not been developed for web-based services handling personal information, and consequently is limited in its ability to describe properties such as data-flow within a web-service.

Note that we are not addressing the problem of malicious service providers: there can be no technological solution that provides protection from providers that intentionally lie about their usage of consumer data. Even source-code analysis techniques are of no avail here: these techniques can ensure that a *given piece of software* satisfies a certain policy, but in a distributed environment, consumers have no authority (or even access) to the computers hosting the service. Therefore, they cannot prevent a malicious service provider from simply switching to a different piece of software that violates these policies. As such, technology cannot prevent malicious providers that misrepresent their terms of usage, or provide incomplete information about it. Indeed, our approach relies on the premise that due to market or societal forces, providers and consumers have already decided to collaborate in order to preserve consumer privacy, and that our approach simply provides a technological basis to facilitate this collaboration. We therefore assume some non-technological means (e.g., legal enforcement) can address the problem of such malicious providers.

1.1 Overview of Approach

In our approach, consumers can express their privacy concerns using *privacy policies*. These policies capture what types of consumer data they are willing to share with which service providers ("principals"). The type of consumer data is identified using *data labels*. For instance, a password may be classified as "secret," a credit-card number as "highly sensitive," telephone number as "sensitive," and the country of residence as "public." It is possible that multiple labels may be associated with the same data, each capturing one aspect about its use. For instance, data regarding a consumer's prescriptions may have a label "health," while a tax data may be labeled "financial." Based on these labels, policies simply state that data with a certain label can (or cannot) be made available to a certain principal. (In practice, other abstractions to simplify policy management may be layered on this simple policy language, but those are not important for this overview.)

When a consumer provides data to a service provider, she wants to ensure that it is used in a manner consistent with her policy. To verify if this will be the case, the consumer requests a *model* of the service, which captures the manner in which the service uses consumer data. These models capture how the information provided to the service can flow to various principals. If the service does not use other services in turn, then there is only one principal involved, namely the service provider itself. If the provider uses other services, the providers of these services will also need to be represented in the model.

A model basically captures the flow of user data to various principals. For instance, it may state that input I_1 is given to principal P_1, while inputs I_2 and I_3 are given to principal P_2. A model may be *incomplete* in that it may not explicitly name all of the principals that it interacts with. This is permitted for two reasons. Some providers (e.g., comparison shopping service provider) may interact with far too many principals that it may not be practical to list them all in a model. Second, the service provider may consider this sort of information to be proprietary, and may not want to share it with the consumer.

When a consumer receives the model, he can check if the model is compatible with their policies. To do this, the consumer labels each of the data items that he is providing as input to the service. (In reality, labels will, in most cases, be generated automatically by a user's application.) In addition, the user provides his/her privacy policy. To perform policy compliance check, the consumer's application uses the model to discover the labels on the data that will be sent to different principals mentioned in the model. Then it checks if the consumer's policy allows data with these labels to be sent to those principals. If the check succeeds, the application forwards the request to the service provider. If the policy is violated, there are several possibilities:

- The user looks for an alternate service provider that satisfies his policies, or
- The user may relax their policy so that the service can be used. In this case, the user is told the manner is which the policy is being violated, e.g., zip code (a

data item with the label "address") is being sent to an Internet merchant through a service that provides comparison shopping, or

- The violation may be forwarded to the provider to see if the service can be provided without violating consumer policy, e.g., the comparison shopping site may be informed that zip code should not be sent to any merchants. The comparison shopping site may still be able to provide most of its services, e.g., provide price quotes from those merchants that do not ask for zip code.

1.2 Distinction from Previous Work.

A unique concern that our work addresses is to resolve incompatibilities between consumer policies and composite service providers' terms of use. Previous works were that addressed consumer privacy were simply concerned with policy enforcement, with no regard for resolving such incompatibilities that do frequently arise in practice.

Another benefit of our approach is that it does not require producers or consumers to fully reveal their models or policies. In particular, producers may not name all the services that they use in turn. They may use a wild-card in the place of a principal name to denote that certain data may be sent to undisclosed principals. In such a case, there will be a violation of the consumer policy regarding that piece of data. Rather than simply aborting the transaction at this point, the consumer can pass on his policies regarding that data to the service provider, who may be able to provide the service by going to only those servers to which the consumer permits this data item to be sent. Note that the consumer has to reveal some additional data in this case, i.e., provide part of its policy to service provider. But this is only the part of the policy that concerns data being sent to undisclosed providers. The rest of the policy is not disclosed to the service provider.

In contrast with previous works, our work provides a "practical" and "framework" for solving this problem. A number of previous research efforts have gone into privacy preservation in software, but all of them have turned out to be largely of theoretical value due to the following reasons:

- Overly conservative models of information flow that, in practice, leads to the conclusion that every piece of software leaks every sensitive information in its outputs, even though in reality, they do not leak any significant information.
- Reliance on source-code analysis techniques, which require access to source code of *all* the services provided by various web sites accessed by a composite service.

Overcoming these limitations requires a fundamental reevaluation of the assumptions made by previous works. By undertaking such an evaluation, we have developed a new approach that makes a judicial combination of trust (on service providers to accurately specify information flows) and verification (to check conformance of these flows with end-user specified privacy policies) to avoid requirement of source-code for remote services provided over the web.

A second key point about our approach is that it is concerned with the development of a *general framework,* rather than a specific end-product that can be integrated with existing web service technologies to yield a solution. To illustrate its practicality and utility, we are implementing it in the context of *Personalized Information Agents (PIAs).* PIAs are intelligent agents that are capable of navigating web sites to access the capabilities provided by the web site without requiring explicit navigation actions by the user. They provide a programmatic interface to access these capabilities. Thus, PIAs can be thought of as a "wrapper" that layers over web sites that are intended for user by humans, and in effect, turn them into web services. This factor enables us to proceed with the implementation without having to be constrained by the limited deployment of *open* web services (as opposed to closed or restricted web services, where the service may be proprietary, for internal use, or may require contractual agreements.)

1.3 Implementation Context

We are primarily concerned with composite web-based services that provide advanced services to end-users based on simpler services provided by multiple providers. We believe that the greatest potential for advanced web-based services lie in the context of such composite services; and that they pose the greatest challenges in terms of privacy preservation.

Illustrative examples of composite web-based services include:

- Cheap, over-the-counter medicine procurement example outlined earlier
- A service that makes a list of used Jaguar cars advertised in New York City area, such that each car is a 1999 or later model, has good safety ratings, and its selling price is less than its Blue Book value.
- An electronics purchase service that integrates research, reviews, comparison shopping and so on into a single, easy-to-use service.
- A travel arrangement service that can, in one step, make all travel arrangements, including airline reservation, hotel accommodation, local transportation arrangements, and more.

Although the component services will be provided by PIAs in our implementation, the overall approach is not tied to the PIA model at all. Indeed, the framework can be applied in more or less a straight-forward way if the PIAs were to be replaced by web services, or vendor-provided software that executes purely on the end-user's computer.

This rest of the paper is organized as follows: Section 2 gives an overview of our framework architecture. Section 3 describes the composition infrastructure. We then discuss our approach for preserving privacy in Section 4 (privacy policies), Section 5 (service models), and Section 6 (privacy policy enforcement).

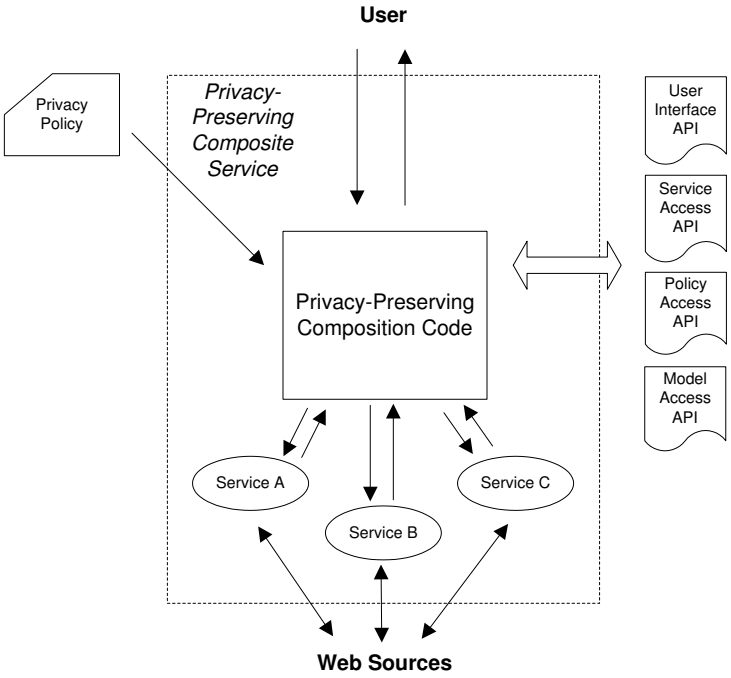

Figure 1: Framework architecture

2. FRAMEWORK ARCHITECTURE

Figure 1 shows the architecture of our framework. The *component services* are represented by ovals in the lower part of the figure. Each such service accepts inputs from its caller, then interacts with other entities such as web sites, and returns requested results to the caller. These services are then glued together using the *composition code*, resulting in a composite service which typically provides richer functionalities. User-defined privacy policies are also enforced in the composite service to preserve users' privacy.

Let us take a concrete example to give an idea about the function of the framework. A detailed description of our techniques employed for service composition and privacy preservation are presented in the following sections.

Suppose that the composite service in Figure 1 is a travel management service, which is composed from a JetBlue air ticket booking service (Service A), a Hotels.com hotel reservation service (Service B), and a MapQuest driving direction service (Service C). Users can use the composite service as a one-stop shop for their travel planning and reservations. To use this service, a user first provides his travel information (e.g. the origin/destination cities) and his/her payment information (e.g. the credit card number) to the composite service. The composite service will invoke Service A with the origin/destination addresses and payment information to book an air ticket; then request Service B to reserve a hotel near the destination address, and execute Service C to retrieve the driving directions from the destination airport to the reserved hotel.

To protect users' privacy in the composite service, the user needs to define his privacy policy to specify his/her own privacy concerns. One example privacy policy might be that both origin/destination addresses and payment information are allowed to be sent to JetBlue and Hotels.com, but the destination address is the only piece of information that is allowed to be sent to MapQuest. Each component service is required to provide a model that describes how its input data is handled by other principals that the service interacts with. For instance, the model of Service A, the JetBlue air ticket booking service, might be described as "origin/destination addresses and payment information are sent to JetBlue". These models are used by the service composition code to verify if the services violate the specified privacy policy before executing these services. Because data can flow from one service to another service in the composition code, a program transformation technique is developed to instrument the composition code to track such information flows as well as to perform policy verification checks.

We should point out that our framework provides a general way of building new services based on services which have already been built. This entails that the composite services themselves can be also used as a component service to construct another service. Our privacy preserving techniques will continue to work.

3. SERVICE COMPOSITION

As the need for web services is growing quickly, rapid and effective web service development techniques become increasingly important. As such a technique, service composition, in which an advanced service can be built by composing a number of smaller services that implement a relatively simpler task, shows great potential by enabling service re-use.

3.1 Composition Language

We shall use some formalism to introduce the composition framework. A natural formalism involves construing every service as a function that maps a set of inputs to a set of outputs. Each service may also interact with a number of external entities such as web sites for accomplishing its functionality. Let $(i_1, i_2, i_3, \ldots, i_n)$denote a set of inputs to an individual service A, and the corresponding outputs be $(o_1, o_2, o_3, \ldots, o_m)$. Each of the individual inputs can come from primitive types such as `integer, real, string,`, or complex types such as `array, record`. Now each composition can be understood as a function that maps the outputs of service A to inputs of service B.

The composition constructs are written in a simple, Java-like imperative language that uses the above-mentioned primitive and complex types. The language includes the following constructs: basic arithmetic, logical expressions, facilities for string matching and editing, loops and procedure calls.

Services are central to composition, and we define APIs for accessing services in the composition code.

- **Service**. Each object of type `Service` represents an component service, and this object can be constructed from a service definition XML file. `Service` defines the following methods:
  - `Service(String srv_fname)`
  - `int setInput(String name, Object value)`
  - `Object getOutput(String name)`
  - `int execute(String out_fname)`

3.2 Composition Example

Before giving the example of composition, we first briefly introduce the WinAgent system, which is used in our implementation to create open wrapper services around existing web sites.

3.2.1 WinAgent System

The WinAgent system [3] is for creating and executing Personal Information Assistants (PIAs) which can automatically extract data of interest from web pages on behalf of a user. More importantly the user can create and execute PIAs using just a web browser. Essentially the user only needs to highlight in the browser examples of data of interest in a web page and the links to be followed and/or forms to be filled to reach this page. From the highlighted examples the system creates an PIA by "learning" navigation and extraction expressions (called navigation map), and stores the navigation map into an XML file. A PIA interpreter can then interpret this navigation map by automatically navigating to the web site, following specified links, filling out forms and extracting the targeted data from the pages as specified. The output of the interpreter is an XML document with attribute names that were supplied by the user when the PIA was created.

Below is the composition code snippet of the travel management service example described in Section 2. The component services are implemented as PIAs created by the WinAgent system.

```
Service jb = new Service(``jetblue_service.xml'');
jb.setInput(``sourceCity'', srcCity);
jb.setInput(``destinationCity'', destCity);
jb.setInput(``creditCardNo'', creditCardNo);
status = jb.execute(``jetblue_out.xml'');
price = jb.getOutput(``price'');
confirmNo = jb.getOutput(``confirmNo'');

Service htl = new Service(``hotels_service.xml'');
htl.setInput(``address'', destCity);
htl.setInput(``creditCardNo'', creditCardNo);
status = htl.execute(``hotels_out.xml'');
hotel = htl.getOutput(``Hotel'');

Service mq = new Serivce(``mapquest_service.xml'');
mq.setInput(``fromAddress'', destCity);
mq.setInput(``toAddress'', hotel);
status = mq.execute(``mapquest_out.xml'');
directions = mq.getOutput(``Directions'');
```

4. PRIVACY POLICIES

A privacy policy describes the privacy requirements of a user by defining constraints on how his data could flow between different entities. Because privacy concerns vary across users and users may treat their privacy policies themselves as private, privacy policies are defined by users and not part of composite services in our framework. Policy specifications are also neutral to services, i.e., they express a user's privacy concern independent of the services that might be executed. We will discuss how to use such policies to preserve users' privacy in Section 6.

4.1 Policy Language

To specify privacy policies, our policy language makes use of an important concept, *label*, which is described as follows.

4.1.1 Labels

In our approach, a piece of user's data, called *an data item* can flow into various entities that are called *principals*. Examples of principals include services or web sites. Both data items and principals are grouped using *labels* to ease policy specification and management.

Labels are classified as *data labels* and *principal labels*, which are used to group data and principals, respectively. Each label is a vector of label attributes, each representing a particular privacy-related aspect of data or principals. Each label attribute can have a finite number of possible values. Consumers are allowed to define any label attributes that can best express their privacy requirements. In our implementation, for example, we define label attributes *sensitivity level* and *information category* for data labels, and label attribute *trust level* for principal labels.

The attribute *sensitivity level* describes how sensitive a piece of data is, and the possible sensitivity levels can be *highly sensitive*, *sensitive*, *less sensitive*, and *public*. For instance, one would think that his credit card number and bank account number are secret; the credit card expiration date or the bank name are sensitive; and data such as which country he lives in are not sensitive. Then these data items can be assigned to sensitivity levels *highly sensitive*, *sensitive*, and *less sensitive*, respectively.

Data items can also be categorized into different informa-

tion categories. One might classify his social security number (SSN) and date-of-birth as personal information, his credit card numbers as financial information, and his residence street names and numbers as address information. This is described by the label attribute *information category*. Figure 2 presents an example of some possible data labels based on sensitivity levels and information categories.

A user may trust different web sites or web services to various extent. A well established on-line retailer might be deemed as "highly trusted", a random web site might be thought as "untrusted", and a small known web site might be treated as "less trusted". The principal label attribute *trust level* is used to express such users' privacy preferences.

Using these label attributes, for example, sensitivity level and information category, we can to express more accurate privacy policies. Suppose a person may think that both his SSN and credit card number are sensitive information. When he is shopping in an on-line store, he may be willing to give his credit card number to the store, but not his SSN number. By classifying SSN and credit card number into different categories, we can differentiate the information flow requirements for these two applications.

A partial order is defined on each label attribute. The partial order can be either natural order based (e.g. *highly sensitive* $\leq$ *sensitive*) or subset based (e.g. {personal} $\subseteq$ {personal, financial}). A complete lattice is defined on the label based on these partial orders on label attributes. Based on the label lattice definition, several operator functions are defined to facilitate inter-label operations, such as computing the lower label for a given label, or computing the least upper bound label of two labels.

4.1.2 Policy Rules

A user's privacy policy consists of a set of policy rules. A policy rule is of the form:

$$(data\ label\ X, principal\ label\ Y, action)$$

which either grants a flow of data with label X to a principal with label Y when *action* is specified as "allow", or restricts such a flow when *action* is "deny". As an example, if a user does not want to release any sensitive financial information to a less trusted site, he can specify a policy rule:

$$(< sensitive, financial >, < less\ trusted >, deny)$$

4.2 High-level Policy Management

Although it is possible for a user to define his privacy policy by directly using the primitive data labels and principal labels, it would be much more desirable that privacy policies can be defined at a higher level, as this can greatly simplify the policy definition process and make the privacy policy management much easier.

One possible way of higher level privacy policy definition is to use *application domains* (e.g. a shipping application, or a banking application). We can first define application domains , then map each application domain to its required data labels. Privacy polices will then be defined in terms of application domains and principal labels. For example, we can define an application domain "shipping", then principals that handle shipping can be allowed to access all the data required for complete the shipping transaction (e.g. all the data belonging to the information category "Address").

5. SERVICE MODELS

The model of a web service is built by the service producer and encodes the data flows among its inputs, outputs, and interacting principals. These models are used by the composite service mainly in three ways:

- The model can be used to verify whether or not the corresponding service respect the user's privacy policy.
- In cases where policy violations are reported, the model can be used to create a projection of the privacy policy and the *policy projection* can be passed along with other inputs to the service. If possible, the service can execute an alternative implementation which does not violate the policy projection passed to it.
- The model can be used to infer the data flows from the inputs of the corresponding service to the outputs of the same service. This is of particular importance in tracking data flows between component services where outputs of one service may be passed as inputs to another service.
- In situations where new services are composed from existing composed services. [1]

5.1 Model Language

A model defines two kinds of data flows which happen in a service: from an input to an output, and from an input to a principal. These data flows can be characterized using the following constraint expressions (here we do not distinguish principals and outputs):

- *Data independence.* To denote that the output Y does not have any dependence on data stored in X (read "Y is independent of X's data"), we use the notation `indep(Y,X)`. Since the model specifies all possible flows within the service, for any input X and output Y, `indep(Y,X)` is by default, true.
- *Exact dependence.* To denote the (exact) nature of the relationship between Y and X, we use the notation `depend(Y,X)`. In cases where more accurate dependence data is available, one can use any expression constructs available for expressing such dependences. For instance Y := X mod 2 expresses the exact relation that Y contains the last bit of X, and Y = X.
- *Quantitative dependence.* To denote that the output Y does contain a certain quantity of data derived from the input X, then the notation *quant(Y,X,b)* is used. The third argument *b* tells how much data contained in Y is derived from X, and it can be quantified in many ways, e.g. the number of bits, or a percentage and so on.

[1] When we use a composite service as a component service to construct another new service, the model of the composite service, which is required by the new service, is obtained through the composition of the models of its component services. We will not discuss how this composition is done in this paper. However, it should be a straight-forward process.

	Highly sensitive	Sensitive	less sensitive
Personal	SSN, DoB	age	name,gender
Address	apt. no.	street no.	city, zip code, state, country
Financial	account no.	routing no.	have checking?
Payment	CC no.	CC exp date	CC type
Health	medical history		height, weight

Figure 2: Data labels based on sensitivity levels and information categories.

5.2 Model Example

We present part of an example model for the JetBlue service in the travel management composite service as follows:

```
[Input]
  srcCity
  destCity
  creditCardNo

[Output]
  price
  confirmNo

[Principal]
  Jetblue

[Model]
  depend(Jetblue, srcCity)
  depend(Jetblue, destCity)
  depend(Jetblue, creditCardNo)
  quant(price, srcCity, 0.5)
  quant(price, destCity, 0.5)
  quant(confirmNo, creditCardNo, 0.3)
```

6. PRIVACY POLICY ENFORCEMENT

6.1 Label Initialization

As we mentioned before, the consumer's privacy policy specification is defined based on labels, and is independent of actual data items and principals in services. So as the first step to enforce the privacy policy for a service, all the consumer input data items and all the involved principals in the service need to be assigned to appropriate labels. This is done before the service execution by a user-agent application to ask the user or automatically query an incrementally maintained labeling database.

During label initialization, we also need to initialize the mapping from flow dependence types in a service to appropriate label operator functions. This mapping is very useful for declassifying data labels during information propagation. Let us assume that a service contains a flow from an input X to an output Y with the dependence quant(0.3), where 0.3 indicates that only small amount information leaked from X to Y. In this case the consumer is willing to declassify Y if X is sensitive. By mapping quant(0.3) to the label operator "lower", Y will be declassified as "public" after executing the service.

This label initialization process is the dynamic part of a consumer's privacy policy, and completes the whole privacy policy specification. By making different label assignments carefully according to services, the privacy policy specification can remain unchanged while the consumer's changing privacy requirements can still be satisfied.

6.2 Privacy Policy Verification

Before a service is executed, its model is retrieved and verified against the privacy policy. First, appropriate labels are assigned to the inputs and interacting principals of the service. The labels are then propagated along the flows defined in the model to principals. Depending the dependence types of the flows, the corresponding label operator functions will be applied to obtain the result labels. These result labels and their associated principal labels will checked against the set of rules specified in the privacy policy. If there is no violations, the service is allowed to execute. Otherwise, conflict resolving mechanisms will be engaged.

The simplest way of resolving policy violations is to suspend the execution of the service, or to relax the privacy policy after the consumer being notified with the violating rules. Alternately a projection of the privacy policy on the service model can be created and passed along with other inputs to the service. In the latter case, the service is trusted in respecting the privacy policy passed to it. The service may still have a successful execution if it can adapt its behavior according to the policy.

6.3 Privacy Preserving in Composite Service

To preserve privacy in composite services where data may flow between various component services, we use a program transformation technique to transform the composition code. The transformation makes use of the models for each component service, and inserts runtime updates and checks into the transformed code to track information flows. More details on the technique and proofs of correctness are available in [11]. A summarized description of the transformation appears below.

- **Variable definitions**. An auxiliary variable is inserted for each program variable to hold the data label of the program variable runtime value. Because principal labels won't change during the service execution, a special global variable is inserted for holding labels of principal which are obtained from the label initializaiton.
- **Statements**. Both explicit and implicit information flows are tracked in the transformed code. To track the *explicit* information flows, when a program variable is updated through an assignment, its label variable is also updated through an inserted assignment in the transformed code. On entering a conditional branch, there is an *implicit* flow of information from the conditional to all the variables assigned in both branches. To compute this set of variables a static analysis is used. The transformation then makes use of the result of this analysis.

The analysis computes the following: for any expression e, $var(e)$ is the set of all variables that are used in the expression. The `if S` denotes a (possibly compound) statement, the set $upd(\mathtt{S})$ denotes all the variables that get assigned in the statement `S`. The analysis computes a conservative upper bound in case a precise estimation of the set of variables that are updated is not possible.

Using the results of the analysis, the program is transformed such that, on entering a conditional branch, the value of the label variable corresponding to the condition expression (associated with enclosing nested conditionals) is stored in a variable called pc_{lb} that is used to track implicit flows. For all assignments that happen inside both branches, the label variables of the left-hand side expressions are updated with the value of the implicit flows, that is obtained by a disjunction with pc_{lb}. Thus, pc_{lb} includes the combined effect of the enclosing conditionals and loops. When a conditional branch is exited, the value of pc_{lb} is restored to its previous value that existed before entering the branch.

- **Services**. Before executing a service, the service model is checked against the privacy policy to prevent unwanted information leaks. The checking process is described in 6.2. When the service execution completes, the labels of the outputs are also computed by propagating the input labels in the service model. These labels are sent along with the outputs to other services whenever those services use the outputs of this service as their inputs.

Below we show an example of transforming part of the travel management service composite code.

```
PrivacyPolicy policy = new
    PrivacyPolicy(``privacy-policy.xml'');
...
UserInterface ui = new UserInterface();
ui.readInput(``creditCardNo'', creditCardNo);
ui.readInputLabel(``creditCardNo'', creditCardNo_lb);
...
Service jb = new Service(``jetblue_service.xml'');
IflowModel jb_model =
    new IflowModel(``jetblue_model.xml'');
...
jb.setInput(``creditCardNo'', creditCardNo);
jb_model.setInput(``creditCardNo'', creditCardNo_lb);

if (jb_model.verifyPolicy(policy) != SUCCESS)
raise PolicyViolationException;
status = jb.execute(``jetblue_out.xml'');

price = jb.getOutput(``price'');
confirmNo = jb.getOutput(``confirmNo'');
price_lb = jb_model.getOutput(``price'');
confirmNo_lb = jb_model.getOutput(``confirmNo'');
```

7. RELATED WORK

Access control based approaches (including *sandboxing* [6] approaches) do prevent information from leaking from a given resource (such as a user's computer or a web site), but they do not address the problem of controlling access to a piece of information once it leaves a given host with the consent of the user. An approach that addresses privacy of information must address the problem of control to be effective. Other approaches such as cryptography, (and associated programs like OpenPGP [7]) have the same effect: they are indispensable for making personal information truly secretive, but do not address the problem of controlling information that is once passed on to another entity.

Anonymity [5] is often another approach taken for privacy protection, where the data relating to a particular user is "anonymized" to be free from any information that can be associated with the user. Anonymity has been of partial success in addressing *passive* threats to privacy, such as those that However, anonymity is of limited use when performing transactions that inherently need disclosure of personal information, such as making airline reservations or making credit card purchases.

The security community has been investigating with program analysis techniques for securing programs from such information leaks [8, 12]. As mentioned in the introduction, these approaches are not particularly suitable in the context of services provided by various web sites, due to two factors. The first their use of overly conservative notions of information flow, that in practice leads to the conclusion that every piece of software leaks every sensitive information in its outputs, even though in reality, they do not leak any significant information. Also, in order to be effective, these approaches need the source code of the services provided by web sites accessed by the composite service, which is difficult to achieve in practice.

Other solutions such as P3P [2] (Platform for Privacy Preferences) have aimed at web resources such as cookies and files. P3P has not been developed for web-based services handling personal information, and consequently is limited in its ability to describe data-flow within a web-service.

Several NSPW papers in the past have presented ideas on information privacy issues. Gates and Slonim [4] presents an approach where owners control access to the individual information by being the sources of any information pertaining to them. While this approach does place tighter holds on information dissemination, it does not address the orthogonal problem of automating privacy policy compatibility/checking of users and the entities they interact with. Ashley, Powers and Schunter [1] presents an approach for enterprises to deal with increasing consumer demands on their personal information privacy. Finally, the inspiration for this project comes from the MCC project, presented in NSPW 2001 [9, 10]. While MCC addressed protection of system resources while running untrusted programs, our current framework addresses user privacy in the context of composite web services. Our research in adopting the MCC paradigm in the area of privacy has yielded the framework and results presented in this paper.

8. CONCLUSIONS

In this paper, we have sketched a framework for preserving privacy in web-based services. In our framework, consumers can have facilities to specify their privacy concerns through use of privacy policies, while service providers express their terms of use (of private data) through models. The com-

patibilities between privacy policies and service models can be verified during the time of service execution. We believe that a judicial combination of trust and verification is needed to address information privacy issues in web services, and the approach presented in this paper is a first step in that direction.

9. REFERENCES

[1] P. Ashley, C. Powers, and M. Schunter. From privacy promises to privacy management: a new approach for enforcing privacy throughout an enterprise. In *New security paradigms Workshop*, 2002.

[2] L. Cranor. *The Platform for Privacy Preferences 1.1 (P3P1.1) Specification*. W3C working draft, July 2004.

[3] A. Gandhre, P. Santhanagopalan, P. Singh, D. Ramavat, I. Ramakrishnan, and H. Davulcu. Creating and managing personal information assistants via a web browser: The WinAgent experience. In *Workshop on Information Integration on the Web*, Toronto, August 2004.

[4] C. Gates and J. Slonim. Owner-controlled information. In *New security paradigms Workshop*, 2003.

[5] I. Goldberg, D. Wagner, and E. Brewer. Privacy enhancing technologies for the internet. In *IEEE COMPCON*, 1997.

[6] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *USENIX Security Symposium*, 1996.

[7] Openpgp proposed standarnd rfc 2440. http://www.ietf.org/rfc/rfc2440.txt.

[8] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1), Jan. 2003.

[9] R. Sekar, C. Ramakrishnan, I. Ramakrishnan, and S. Smolka. Model carrying code: A new paradigm for mobile code security. In *New Security Paradigms Workshop (NSPW)*, 2001.

[10] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model carrying code: A practical approach for safe execution of untrusted applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[11] V.N. Venkatakrishnan. *Enforcement Techniques for Expressive Security Policies*. PhD thesis, Stony Brook University, 2004.

[12] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security (JCS)*, 4(3):167–187, 1996.

APPENDIX

A. FRAMEWORK XML AND API DEFINITIONS

In this appendix, we provide a design of XML and API definitions that can be used to specify privacy policies and service models, and facilitate the privacy policy enforcement.

A.1 Label

Both consumer data and service providers ("principals") are grouped using *labels* to ease policy specification and management.

A.1.1 Label XML Definition

Labels are classified as *data labels* and *principal labels*, which are used to group data and principals, respectively. Each label is a vector of label attributes, each representing a particular privacy-related aspect of data/principals (e.g. sensitivity level, or information category). Each label attribute can have a finite number of possible values (e.g. secret, sensitive, public), and a partial order (either natural-order-based or subset-based) is defined over these attribute values.

The definitions of label attributes and the composition of the label vectors are specified by consumers as the first part of an XML policy specification, *label definition* or **labeldef**:

- **labelattr**: Defines a label attribute.
  - **name**: Name of the label attribute (e.g. sensitivity_level).
  - **values**: A list of possible label **value**'s (e.g. *secret, sensitive, public*).
  - **ordering**: Defines the **ordertype** and partial order of label values (**order**). Two kinds of **ordertype** can be specified: *natural-order*-based ordering and *subset*-based ordering.
  - **opmapping**: Defines the mapping (**opmap**) between dependence types (**dependence**, used in service models) and label operators (**labelop**). There are five predefined label operators:
    - *id(l)*: the identity operator and always returns *l*
    - *lower(l)*: returns a smaller/lower label value than *l*
    - *upper(l)*: returns a bigger/higher label value than *l*
    - *glb(l1, l2)*: return the greatest-lower-bound label value of *l1* and *l2*
    - *lub(l1, l2)*: return the least-upper-bound label value of *l1* and *l2*

    The label results after applying the above operators on given label values also depend on the partial order defined in **ordering**.
- **labelvector**: Specifies which label attributes used in a label vector.

A.1.2 Label Java API

Data labels and principal labels are represented as objects of **DataLabel** and **PrincipalLabel**, which are simply container classes (i.e. they are only used to store any kinds of user-defined label vectors). Actual operations on labels are defined in **DataLabelOp** and **PrincipalLabelOp**, which will be instantiated from the above XML label definition. Both **DataLabelOp** and **PrincipalLabelOp** are data members of **PrivacyPolicy** due to the fact that the label definition is part of privacy policy specification.

The following APIs are defined in **DataLabelOp** (Similar APIs are defined in **PrincipalLabelOp**, too):

- **DataLabel lower(DataLabel l)**: Returns a label lower than *l*
- **DataLabel upper(DataLabel l)**: Returns a label higher than *l*
- **DataLabel leastUpperBound(DataLabel l1, DataLabel l2)**: Returns the least upper bound of labels *l1* and *l2*.
- **DataLabel greatestLowerBound(DataLabel l1, DataLabel l2)**: Returns the greatest lower bound of labels *l1* and *l2*.
- **DataLabel dependant(DataLabel l, Dependence d)**: Returns the return label after applying to *l* the operation corresponding to the dependence type *d*.

A.1.3 Label Initialization

As part of consumer's privacy policy, initially labels need to be assigned to input data and involved principals in a service. This is done in the beginning of service execution by asking the user or querying an incrementally maintained grouping database. After initialization, data label values are stored in the data label variables associated with each program variable, while principal label values are kept in a database **PrincipalLabelDB** for later queries (reason: principal labels won't change during the execution).

The following APIs are defined in **PrincipalLabelDB**:

- **int setPrincipalLabel(String name, PrincipalLabel l)**: Set the label of the principal *name* to *l*.
- **PrincipalLabel getPrincipalLabel(String name)**: Returns the label of the principal *name*.

A.2 Privacy Policy

Consumers can express their privacy concerns using *privacy policies*. These policies capture what types of consumer data they are willing to share with which service providers. Policy definitions are based on *labels*.

A.2.1 Policy XML Definition

There are two parts in an XML policy file (**privacypolicy** as the root element): *label definitions* (**labeldef**), as described previously, and *policy rules* (**policydef**).

In policy rule specification, each **rule** either grants a flow of data with label X to a principal with label Y (e.g. *((secret, payment), (trusted), allow)*), or restricts such a flow (e.g. *((secret, payment), (untrusted), deny)*).

A.2.2 Policy Java API

A privacy policy is represented as an object of **PrivacyPolicy**. The following APIs are defined in PrivacyPolicy:

- **PrivacyPolicy(String xml_file)**: Constructs a PrivacyPolicy object from the XML policy specification file *xml_file*.
- **int checkPolicy(IflowModel model)**: Checks whether or not the given service model respects this privacy policy.
- **PrivacyPolicy projectPolicy(IflowModel model)**: Returns part of this privacy policy that only concerns data being sent to the service represented by *model*.
- **DataLabelOp getDataLabelOp()**: Returns the DataLabelOp object.
- **PrincipalLabelOp getPrincipalLabelOp()**: Returns the PrincipalLabelOp object.

A.3 Service Model

A service model captures the manner in which the service uses consumer data. Specifically, these models capture how the information provided to the service can flow to various principals.

A.3.1 Model XML Definition

An XML service model (**iflowmodel** as the root element) defines the inputs and outputs of the service, the principals which the service interacts with, and the information flows from inputs to principals and outputs.

- **inputdef**: Defines a list of **input**'s. Each input is described by its name, datatype, and other attributes.
- **outputdef**: Defines a list of **output**'s. Each output is described by its name, datatype, and other attributes.
- **principaldef**: Defines a list of **principal**'s. Each principal is described by its name, URL, and other attributes.
- **flowdef**: Defines a list of **flow**'s. Each flow is described by its source, destination, the type of flow (either *InputOutput* or *InputPrincipal*), and the type of dependence. The source of a flow is always an input, while the destination of a flow can be an output or a principal. Two types of dependence can be specified:
  - **exact**: Defines an exact dependence between the flow source and destination. (e.g. an input is directly sent to a principal, or assigned to an output).
  - **quant(n)**: Defines a quantitative dependence between the flow source and destination, where n denotes the percentage of the source information sent to the destination.

A.3.2 Model Java API

A service model is represented as an object of class **IflowModel**. The following APIs are defined:

- **IflowModel(String xml_file)**: Constructs an IflowModel object from the XML model specification file *xml_file*.
- **int setInputLabel(String name, DataLabel label)**: Set the label of the input identified by *name* to *label*.
- **DataLabel getOutputLabel(String name)**: Get the label value of the output identified by *name*.
- **Vector getAllPrincipals()**: Returns all principals defined in the model. It is useful for **projectPolicy()** in **PrivacyPolicy**.
- **int setPrincipalLabel(String name, PrincipalLabel label)**: Set the label of the principal identified by *name* to *label*. It is useful for **checkPolicy()** in **PrivacyPolicy**.

A.4 Transformation

Individual, simpler services can be "glued" together as a composite service using composition code. To ensure that the composite service preserves privacy, the composition code is transformed to insert code for information flow tracking (ie. label propagation) and privacy policy checking.

- **Variable definitions**.
  - A PrivacyPolicy variable is inserted to store the user's privacy policy (defined in a XML file).
  - An IflowModel variable is inserted for each component service to store its information flow model (defined in a XML file).
  - An auxiliary variable is inserted for each program variable to hold its runtime data label.
  - A PrincipalLabelDB variables is inserted to hold labels of all principals used by the composite service.
- **Statements**. Both *explicit* and *implicit* information flows need to be tracked and data labels are updated accordingly.
- **Services**.
  - *Service inputs*. When passing variables as inputs to a service, the labels of these variables are also passed as inputs to the model of the service. The principals involved in the service are labelled using corresponding labels in **PrincipalLabelDB**.
  - *Service invocations*. Before executing a service, the model of the service is checked against the privacy policy to prevent unwanted information flows.
  - *Service outputs*. After the the execution of a service the labels of the outputs are updated by querying the service model.