Controlling Data Flow with a Policy-Based Programming Language for the Web

Conference Paper · October 2013

DOI: 10.1007/978-3-642-41488-6_15

Soha Hussein
University of Minnesota Twin Cities

Controlling Data Flow with a Policy-Based Programming Language for the Web

Thierry Sans[1], Iliano Cervesato[1], and Soha Hussein[1]

Carnegie Mellon University
tsans@qatar.cmu.edu, iliano@cmu.edu, sohah@qatar.cmu.edu

Abstract. It has become increasingly easy to write Web applications and other distributed programs by orchestrating invocations to remote third-party services. Increasingly, these third-party services themselves invoke other services and so on, making it difficult for the original application developer to anticipate where his/her data will end up. This may lead to privacy breaches or contractual violations. In this paper, we explore a simple distributed programming language that allows a web service provider to infer automatically where user data will travel to, and the developer to impose statically-checkable constraints on acceptable routes. For example, this may provide confidence that company data will not flow to a competitor, or that privacy-sensitive data goes through an anonymizer before being sent further out.

1 Introduction

Web-based applications (*webapps*) are networked applications that use technologies that emerged from the Web. They range from simple browser-centric web pages to rich Internet applications such as Google Docs all the way to browserless server-to-server web services. In all cases, a client invokes a remote computation and receives a result over the HTTP protocol. To develop webapps, programmers often use third-party web services as building blocks. These third-party services provide specific data and/or processing functionalities that can be accessed through a web API. Using these services is faster, more reliable, and has lower cost than developing an in-house solution from the grounds up. Examples abound in practice: data visualization services such as the Google Maps API to embed a map into a webpage with custom overlay; web analytics services to measure and understand how web services and data are being used; advertising services to embed custom ads into a webpage; credit card verification services to verify the authenticity of a credit card number for an e-commerce website; and customer relationship management services (e-CRM) to externalize the management of customers, just to cite a few.

From the service consumer perspective, even a trusted service is an opaque computation that gives little insight about what it is doing with the data sent to it. In particular, it may outsource part of the processing to a third-party web service without the consumer being aware of it. This could raise confidentiality or privacy concerns if this data is sensitive and it is forwarded to untrusted hosts.

Conventional web service security is often more concerned with constraining access to services than about controlling what these services do with the data.

In practice, when a reputable site offers a web service, its terms-of-use agreement should mention the third-parties involved. However, consumers rarely read these agreements (although they should), and providers may forget to update them when they change third-party services. Furthermore, even when the agreement mentions that the application uses an external service, in most cases it does not list what third-party web services this service may use. In other words, we may become aware of just the first few links in a possibly long chain of services.

These practices do not scale. An application provider should have an automated way to disclose the service locations where user data, or data derived from it, may end up. Similarly an application developer should be able to specify which nodes are deemed acceptable recipients of this data, and to check whether these two lists are compatible. This program has similarities with the goals of P3P [6]: formalize the handshake between the user and the provider of a service. It targets however the flow of data beyond the service provider, and is only concerned with where the data goes rather than finer aspects of privacy.

In this paper, we approach this issue by devising a small web programming language, QWeSST$^{\varphi}$, that statically infers an approximation of the flow of data through web service calls. This allows a service provider to advertise the nodes that user data, or data derived from it, may go through as a call is serviced. More precisely, QWeSST$^{\varphi}$ infers structured paths that user data may traverse. Our language also allows the user to instrument remote service calls with a policy that specifies acceptable paths for the input data to this service. If the policy does not permit the advertised flow, the program will not typecheck. We show that these simple ingredients support useful data flow specifications, some quite complex such as a form of distributed Chinese wall policy. This study is however largely foundational and culminates in establishing type safety for our language.

It should be noted that our goal is not to prevent a malicious service provider from deviating from its advertised data flow—once user data has left the local host, no guarantee can be provided short of relying on trusted computing techniques [10], which are orthogonal to the concerns of this paper. The goal is instead to mechanize the mutual understanding and mutual agreement between the service consumer and the service provider. For instance, the service consumer may express that it does not want supplied data to be forwarded to a certain host. If the service provider does not list that host among the advertised third-party services it uses, the data flow satisfies the service consumer policy and the service call can be executed.

The paper is structured as follows: we introduce a vanilla web language in Section 2 and use it to illustrate common data flows and desired policies on them in Section 3. We outline our model for data flow and a policy language for it in Sections 4 and 5, respectively, and apply them to some examples in Section 6. We incorporate flow tracking and policies in our language, define its formal semantics and show its type safety in Section 7. We review related efforts in Section 8 and outline directions of future work in Section 9.

2 The Base Language

The starting point of our investigation will be a fragment of QWeSST, a simple programming language for the Web [14,15]. This fragment, which we call QWeSST$^-$, is a basic functional language extended with primitives to model remote procedure call (i.e., web services). QWeSST embraces a model of networked computation consisting of a fixed but arbitrary number of *hosts*, denoted w possibly subscripted (we also call them *nodes*). These hosts are capable of computation and are all equal, in the sense that we do not a priori classify them as clients or servers. They communicate exclusively through web services and, just like we normally view the Web, we assume that every node can invoke services from every other node that publishes them.

A web service is an expression of type $\sigma \rightsquigarrow \tau$: it represents a remote function located at host w that accepts arguments of type σ and returns results of type τ. A server w creates a service by evaluating the expression $\mathsf{publish}\ x : \sigma.e$ where e is the computation performed by the service when supplied a value for the formal argument x. This evaluates to a URL w/u, where u is a unique identifier for this service. Once created, a client w' can invoke this web service by calling its URL with an argument of the appropriate type. This is achieved by means of the construct $\mathsf{call}\ e_1\ \mathsf{with}\ e_2$ which is akin to function application. It calls the URL e_1 by moving the value v_2 of the argument e_2 to w, which applies e to v_2 and moves the result back to the client w'.

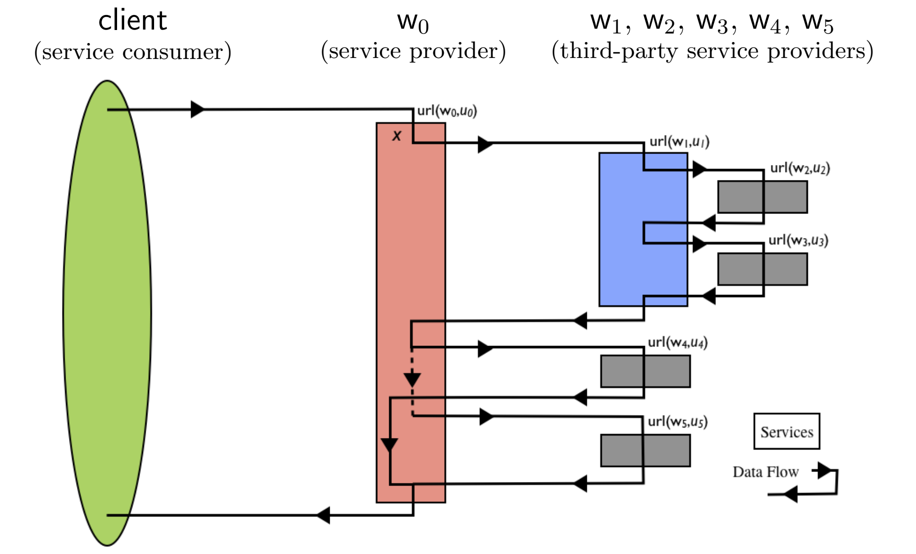
QWeSST$^-$ also includes function, unit and pair types with their usual constructors and destructors—mainly to make our examples more interesting. For technical reasons, we restrict the argument of functions and services to non-functional *base types*. Altogether, the syntax of QWeSST$^-$ is given by the following grammar:

$$
\begin{array}{lll}
\text{Base types} & \sigma ::= \mathsf{unit} \mid \sigma \times \sigma' \\
\text{Types} & \tau ::= \mathsf{unit} \mid \tau \times \tau' \mid \sigma \to \tau \mid \sigma \rightsquigarrow \tau \\
\text{Expressions} & e ::= x \mid \lambda x : \tau.\, e \mid e_1\ e_2 \mid \langle e_1, e_2 \rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid () \\
 & \qquad \mid \mathsf{w}/u \mid \mathsf{publish}\ x : \tau.e \mid \mathsf{call}\ e_1\ \mathsf{with}\ e_2 \mid \textsf{\textit{expect}}\ e\ \textsf{\textit{from}}\ \mathsf{w}
\end{array}
$$

Here, x ranges over variables and u over service identifiers. As usual, we identify terms that differ only by the name of their bound variables and write $[e/x]e'$ for the capture-avoiding substitution of e for x in the expression e'. Service identifiers are never substituted. The expression $\textsf{\textit{expect}}\ e\ \textsf{\textit{from}}\ \mathsf{w}$ is used internally during evaluation and is not available to the programmer. It essentially models a client's waiting for the result of a web service call.

3 Motivations and Approach

In this section, we study an example of a sequence of web service calls, trace its data flow and describe some possible constraints a service consumer may want to impose on this flow. As we go along, we introduce some terminology that we use in the rest of the paper.



```
                      client
let  $e = ()$
in  call $w_0/u_0$ with $e$
end
```
$\hookrightarrow$ $\langle(),()\rangle$: unit $\times$ unit

```
                                 w0
publish $x$ : unit.
 let   $x' =$ call $w_1/u_1$ with $x$
       $x'' =$ call $w_4/u_4$ with $x'$
       $x''' =$ call $w_5/u_5$ with $x'$
 in  $\langle x'', x''' \rangle$
 end
```
$\hookrightarrow$ w_0/u_0 : unit$\rightsquigarrow$unit $\times$ unit

```
                          w1
publish $x$ : unit.
 let $x' =$ call $w_2/u_2$ with $x$
 in  call $w_3/u_3$ with $x'$
 end
```
$\hookrightarrow$ w_1/u_1 : unit$\rightsquigarrow$unit

Fig. 1. Data Flow Example

Our scenario, shown in Figure 1, consists of a service consumer (client, represented as an oval), a service provider w_0, and five additional nodes w_1 to w_5 (drawn as rectangular boxes). The consumer invokes service w_0/u_0 at w_0 with some data v. In order to provide this service, w_0 outsources subcomputations by invoking third-party services w_1/u_1 on w_1, w_4/u_4 on w_4, and w_5/u_5 on w_5. Node w_1 delegates aspects of its subcomputation to nodes w_2 and w_3. The edges describe the flow of input v as it travels through these various nodes. Arrows pointing to the right indicate the flow of data as a service is called. This service uses this data to compute an output value which is returned to the caller through the arrows pointing to the left. We call this output *derivative data* as it may depend on the value input to the service.

Figure 1 illustrates three types of flow, which corresponds to different ways services can be combined.

– Simply invoking a service on a node determines a *service flow*. Here, client, w_0 and w_1 each have service flows. Note that these flows can be cascaded.

– We have a *sequential flow* when a node invokes a service with the result obtained from calling a prior service. For example, servicing w_1/u_1 sequentializes the flows of w_2 and w_3.
– Finally, a *parallel flow* is obtained when calling two services with identical input and combining their results. This is indicated using the dotted line in Figure 1: node w_1 invokes services w_4/u_4 and w_5/u_5 with the same input, and then combines their output before returning a result to client.

The boxes underneath the representations of client, of w_0 and of the other nodes shows examples of QWeSST$^-$ code that could determine this flow. For readability, we rely on an ML-like let construct, where as usual $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$ is understood as $(\lambda x\!:\!\sigma.\, e_2)\ e_1$ for an appropriate type σ. The code for client simply calls w_0/u_0 with some expression, here $()$. The code for w_0/u_0 invokes w_1/u_1 with its input obtaining output x', then calls w_4/u_4 and w_5/u_5 with x' obtaining outputs x'' and x''' respectively and returning them as a pair. For the sake of illustration, we wrap this code around publish to show how w_0 would create it—when evaluated this expression will evaluate to the URL w_0/u_0 invoked by client (the actual URL is passed to client out of band). The code for w_1/u_1 is shown on the right. The code for the other nodes is omitted.

As a service consumer, client has no way to know that w_0/u_0 makes use of third-party web services. While node w_0 may be trusted, some of the nodes among w_1 to w_5 may belong to competitors, in which case the above flow raises confidentiality or privacy concerns if w_0/u_0 is invoked with sensitive data. Indeed, client will not want w_0 to forward (or leak) this data to untrusted service providers.

The rest of this paper will not only provide a way for w_0 to automatically report on the flows client's data will be exposed to, but also to allow a service consumer to instrument service invocations with data flow policies. These policies will define acceptable paths for the input data sent to a service and will be checked statically against the flows advertised by the service provider. For instance, here are two behaviors that client will be able to express as policies:

– The data e submitted to w_0/u_0 cannot be forwarded to node w_3 (which may be a competitor). This is a simple confidentiality policy.
– The data e submitted to w_0/u_0 can be forwarded to w_3 only if it first goes through node w_2. This node may provide well-known services that anonymize or sanitize data, thereby making it acceptable for wider distribution (even to the competitor w_3). This policy incorporates elements of privacy.

In this paper, we will do the following:

– Define a model for data flow and provide an automated way to disclose the data flow incurred by a service as an annotation to its type. Specifically, we will extend the type system of QWeSST$^-$ so that services report the flow of their input: the type of services will now assume the form $\sigma(\!|\mu|\!)\rightsquigarrow\tau$ where μ is a conservative approximation of the data flow of this service. This annotation is not input by the user, but inferred by the type checker.

We similarly instrument other constructs that bind variables, for example functions.
- Define a policy language so that a developer can attach a policy to a value passed as input of a service call. In particular, we update QWeSST$^-$'s service invocation operator of $\mathsf{call}\ e_1\ \mathsf{with}\ e_2$ to $\mathsf{call}\ e_1\ \mathsf{with}\ e_2[\![\rho]\!]$ where ρ is a policy defining constraints on acceptable path for the value e_2 sent to the service e_1. Policies are specified by the user.
- Finally, extend the typing semantics of QWeSST$^-$ to verify statically that the policy satisfies the advertised data flow. Policy verification takes place during typechecking. If a violation is detected, concrete actions could range from raising an error to issuing a warning.

4 Modeling Data Flow

We want to model data flow and capture it in QWeSST$^-$'s types. This will enable service providers to advertise the type and the data flow of the services they offer.

Our notion of data flow closely mimics the modes of composition discussed in Section 3. In particular, we provide operators to express service, sequential and parallel flows. Note that other choices are possible: our notion of flow could have been simply the set of nodes visited during an execution, or it may have reported minute details of how the data is transformed by the various services. Our choice strikes a balance between expressiveness and the ability to compute a meaningful approximation statically.

Our flow language is given by the following grammar:

$$\begin{array}{ll} \text{Possible flows} & \underline{\mu} ::= \_ \mid \mu \\ \text{Actual flows} & \mu ::= \bullet \mid \mathsf{w} \succ \mu \mid \mu\ ;\ \mu' \mid \mu \,\|\, \mu' \end{array}$$

where the various operators have the following meaning:

- A *local flow*, written $\bullet$, indicates that a service does not use any third-party service. It is read "here".
- A *service flow*, written $\mathsf{w} \succ \mu$, indicates that a service uses a third-party web service provided by w that itself has flow μ. It is read "will go to w and then do μ". It can be interpreted as a modal operator indexed by a host.
- A *sequential flow*, written $\mu\ ;\ \mu'$, happens when a service uses a derivative value from μ in μ'. It can be read as "first μ and then μ'".
- A *parallel flow*, written $\mu \,\|\, \mu'$, is when a service uses a value for two independent calculations μ and μ'. It is read "μ and μ' at once".

In addition to these actual flows, a variable x that is never used in a service $x.e$ (i.e., such that x does not appear free in e) is given flow "_" (*no flow*).

We associate the flow of data inferred from a service to its type, which will assume the form $\sigma(\!|\underline{\mu}|\!)\!\rightsquigarrow\!\tau$. Before showing the flow expression corresponding to our example from Section 3, it is convenient to introduce some abbreviations.

We write w for the flow $\mathsf{w} \succ \bullet$, which goes to node w and nowhere further. It will also be convenient to consider ; and $\|$ as associative operators with $\bullet$ as their identity.

The overall data flow observed in our example in Figure 1 is captured by the expression $\mu_0 = \mathsf{w}_0 \succ ((\mathsf{w}_1 \succ (\mathsf{w}_2 \ ; \ \mathsf{w}_3)) \ ; \ (\mathsf{w}_4 \,\|\, \mathsf{w}_5))$. This is the flow advertised by service w_0/u_0, which has therefore type $\mathsf{unit}(\!|\mu_0|\!)\rightsquigarrow(\mathsf{unit} \times \mathsf{unit})$. Notice that each service in that figure advertises some flow. For example, the annotated type of w_1/u_1 is $\mathsf{unit}(\!|\mathsf{w}_1 \succ (\mathsf{w}_2 \ ; \ \mathsf{w}_3)|\!)\rightsquigarrow\mathsf{unit}$.

5 The Policy Language

We now define a policy language that will allow a service consumer to control the data flow of a value sent to a service. The goal is to upgrade QWeSST$^-$'s web service call operator to the form $\mathsf{call}\ e_1\ \mathsf{with}\ e_2[\![\rho]\!]$ where the value e_2 is protected with the data flow policy ρ. This policy ρ constrains the acceptable paths that the value e_2 can take when sent to the remote service e_1. Assuming that e_1 has type $\sigma(\!|\mu|\!)\rightsquigarrow\tau$, the policy ρ shall satisfy the data flow μ advertised in the type of e_1, a constraint that we write $\mu \models \rho$. This will allow us to validate flows statically in Section 7.

This policy language intersperses operators that mimic data flows with traditional Boolean connectives:

$$\text{Policies} \quad \rho ::= \top \mid \bot \mid \neg\rho \mid \rho_1 \wedge \rho_2 \mid \rho_1 \vee \rho_2 \mid \bullet \mid \mathsf{w} \succ \rho \mid \rho_1 \ ; \ \rho_2$$

The meaning of these policy expressions is given by the data flows that satisfy them. This is specified by the judgment $\mu \models \rho$, seen earlier, where μ is a flow and ρ a policy. It is defined as follows:

(1) $\mu \models \top$		(5) $\bullet \models \bullet$	
(2) $\mu \models \neg\rho$	if $\mu \not\models \rho$	(6) $\mathsf{w} \succ \mu \models \mathsf{w} \succ \rho$	if $\mu \models \rho$
(3) $\mu \models \rho \wedge \rho'$	if $\mu \models \rho$ and $\mu \models \rho'$	(7) $\mu \ ; \ \mu' \models \rho \ ; \ \rho'$	if $\mu \models \rho$ and $\mu' \models \rho'$
(4) $\mu \models \rho \vee \rho'$	if $\mu \models \rho$ or $\mu \models \rho'$	(8) $\mu \,\|\, \mu' \models \rho$	if $\mu \models \rho$ and $\mu' \models \rho$

Additionally, $\_ \models \rho$. Here, (1) states that $\top$ is the maximally permissive policy, which is satisfied by every data flow. Dually, $\bot$ is the unsatisfiable policy, as witnessed by the absence of an entry for $\mu \models \bot$. The remaining Boolean operators have a standard interpretation. In particular, $\neg\rho$ complements policy ρ. The remaining operators are more interesting: items (5–7) in this definition state that $\bullet$, $\_ \succ \_$ and $\_ \ ; \ \_$ strictly model local, service and sequential flows, respectively. Although we could have defined a policy counterpart of the parallel flow operator $\_ \,\|\, \_$, we chose in (8) to have parallel flows obey the same policy. The alternative, supporting a policy constructor of the form $\rho \,\|\, \rho'$, seemed artificial as a parallel flow naturally emerges from evaluating expressions in parallel, and therefore imposing an order on the constituent flows does not seem appropriate.

The above satisfiability judgment is consistent (we cannot have both $\mu \models \rho$ and $\mu \not\models \rho$), complete (either $\mu \models \rho$ or $\mu \not\models \rho$), and decidable.

Because the non-Boolean policy operators strictly track the corresponding notions of data flow, it is convenient to introduce a few derived operators as syntactic sugar in order to simplify writing actual policies. In particular, we will permit the following operators, where ws is a set of nodes:

- $\rho_1 \rightarrow \rho_2 \triangleq \neg\rho_1 \vee \rho_2$
- $\mathit{ws} \succ \rho \triangleq \bigvee_{\mathsf{w}_i \in \mathit{ws}} \mathsf{w}_i \succ \rho$
- $\mathit{ws} \succ^\star \rho \triangleq \mathit{ws} \succ (\mathit{ws} \succ^\star \rho) \vee (\neg(\mathit{ws} \succ \bot) \wedge \rho)$
- $\rho_1 \,;^\star\, \rho_2 \triangleq \rho_1 \,;\, (\rho_1 \,;^\star\, \rho_2) \vee (\neg\rho_1 \wedge \rho_2)$
- $\mathit{ws} \succ^? \rho \triangleq \mathit{ws} \succ (\rho \vee \bullet)$
- $\rho_1 \,;^?\, \rho_2 \triangleq \rho_1 \,;\, (\rho_2 \vee \bullet)$

Some of these derived operators deserve an explanation:

- The policy $\mathit{ws} \succ \rho$ simply specifies that data can flow to any node in the set ws and then continue as ρ. It is a simple generalization of the service flow policy $\mathsf{w} \succ \rho$. We will often use it in situations where ws is the complement $\overline{\mathit{ws}'}$ of some set ws'.
- $\mathit{ws} \succ^\star \rho$ and $\rho_1 \,;^\star\, \rho_2$ are the *iterated forms* of policies $\mathit{ws} \succ \rho$ and $\rho_1 \,;\, \rho_2$, respectively. The former describes a policy that allows nested service calls as long as they involve only nodes in ws, otherwise ρ applies. The latter specifies a policy corresponding to an arbitrary sequential composition of flows that satisfy ρ_1 followed by a flow that satisfies ρ_2.
- The forms $\mathit{ws} \succ^? \rho'$ and $\rho \,;^?\, \rho'$ describe policies where the policy ρ' is optional. Specifically, $\mathit{ws} \succ^? \rho'$ specifies a flow that, once it has gone to a node among ws, can either stay there or continue as ρ'. Similarly, $\rho \,;^?\, \rho'$ shall behave as ρ and then can either continue as the sequential flow ρ' or stop there.

As in the case of flows, we write w for the policy $\mathsf{w} \succ \bullet$ and generalize it to sets of nodes, writing ws for $\mathit{ws} \succ^\star \bullet$. Similarly, we simplify $\rho \,;\, \bullet$ and $\bullet \,;\, \rho$ as ρ, except for emphasis. Observe that $\bullet$ ("here") is a very different policy from $\top$ ("anywhere") and from $\bot$ ("nowhere").

6 Policy Examples

We will now examine several examples of policies based on the scenario introduced in Section 3. In each situation, we will define a policy ρ that client can use to constrain the acceptable flows during an invocation of w_0/u_0. In particular the call to this service will assume the form $\mathsf{call}\ \mathsf{w}_0/u_0\ \mathsf{with}\ e[\![\rho]\!]$ for the various policies ρ we will consider. As we write them, we will make abundant use of the derived policy expressions just presented. Recall that each of these policies will be checked against the flow advertised in the type of w_0/u_0, which was:

$$\mathsf{unit}(\!|\mathsf{w}_0 \succ ((\mathsf{w}_1 \succ (\mathsf{w}_2 \,;\, \mathsf{w}_3)) \,;\, (\mathsf{w}_4 \parallel \mathsf{w}_5))|\!)\rightsquigarrow\mathsf{unit}$$

Here are these polices ρ:

- $\rho = \mathsf{w}_0 \succ ((\mathsf{w}_1 \succ (\mathsf{w}_2 \ ; \ \mathsf{w}_3)) \ ; \ \{\mathsf{w}_4, \mathsf{w}_5\})$. Here client defines a policy that corresponds to the exact flow of w_0/u_0. If client has prior knowledge of the data flow of this service, this implements a form of least privilege.
- $\rho = \mathsf{w}_0 \succ ((\mathsf{w}_1 \succ ((\mathsf{w}_2 \succ \top) \ ; \ \top)) \ ; \ \{\mathsf{w}_4, \mathsf{w}_5\})$. Here client trusts w_2 calling any service as needed. Furthermore w_1 can do anything with a result coming from w_2. But upon coming back from w_1, derivative data shall go to either w_4 or w_5.
- $\rho = \mathsf{w}_0 \succ ((\mathsf{w}_1 \succ \top) \ ;^{?} \ \{\mathsf{w}_4, \mathsf{w}_5\})$. Here client trusts w_1 calling any services as needed, and w_0 may send the result coming from w_1 to w_4 and/or w_5 if needed.
- $\rho = \mathsf{w}_0 \succ ((\{\mathsf{w}_1, \mathsf{w}_4, \mathsf{w}_5\} \succ \top) \ ;^{\star} \ \bot)$. Here client trusts w_1, w_4 and w_5 calling any service as needed. However, services from other hosts cannot be called from w_0.
- $\rho = (\{\mathsf{w}_0, \mathsf{w}_1, \mathsf{w}_2, \mathsf{w}_3, \mathsf{w}_4, \mathsf{w}_5\} \succ^{\star} \bot) \ ;^{\star} \ \bot$. Here client trusts w_0, w_1, w_2, w_3, w_4 and w_5 only. They can call each other and pass derivative results around. However, services from any other nodes shall not be involved.
- $\rho = \overline{\overline{\{\mathsf{w}_6\}}} = (\overline{\{\mathsf{w}_6\}} \succ^{\star} \bot) \ ;^{\star} \ \bot$. Here client trusts any node except w_6. (We will make further references to this particular policy in the remainder of the paper, and write $\overline{\overline{ws}}$ to denote the policy that allows anything except going to nodes in ws.)
- Next, client does not allow data to go to w_5 except if it is a derivative value that went through w_2. First, we define a policy ρ_{w_2} that allows a value to be sent anywhere if it is a derivative value from w_2. Second, we define ρ to allow arbitrary paths as long as either w_5 is not involved or the value goes through a flow specified by ρ_{w_2}.

$$\begin{aligned} \rho_{\mathsf{w}_2} &= (\mathsf{w}_2 \succ \top) \ ;^{?} \ \top \\ \rho &= \overline{\overline{\{\mathsf{w}_5\}}} \vee ((\overline{\{\mathsf{w}_2, \mathsf{w}_5\}} \succ^{\star} \rho_{\mathsf{w}_2}) \ ;^{\star} \ \top) \end{aligned}$$

- Finally, we model client wanting to have data flow isolation between w_4 and w_5 à la *Chinese wall security policy*. This means that as soon as the value is sent to w_4 any nested service call or further composition should not involve w_5 and vice versa. Just as in the previous example, we first define a policy ρ_{w_4} where a value can be sent anywhere except to w_5 as long as it is a derivative value from w_4. We define ρ_{w_5} similarly. Then, ρ allows any data flow that does not go through either w_4 or w_5, or if the value goes through one of the two flows specified by ρ_{w_4} and ρ_{w_5}.

$$\begin{aligned} \rho_{\mathsf{w}_4} &= (\mathsf{w}_4 \succ \overline{\overline{\{\mathsf{w}_5\}}}) \ ;^{?} \ \overline{\overline{\{\mathsf{w}_5\}}} \\ \rho_{\mathsf{w}_5} &= (\mathsf{w}_5 \succ \overline{\overline{\{\mathsf{w}_4\}}}) \ ;^{?} \ \overline{\overline{\{\mathsf{w}_4\}}} \\ \rho &= \overline{\overline{\{\mathsf{w}_4, \mathsf{w}_5\}}} \ \vee \ ((\overline{\{\mathsf{w}_4, \mathsf{w}_5\}} \succ^{\star} \rho_{\mathsf{w}_4}) \ ;^{\star} \ \overline{\overline{\{\mathsf{w}_5\}}}) \\ & \qquad\qquad\quad \vee \ ((\overline{\{\mathsf{w}_4, \mathsf{w}_5\}} \succ^{\star} \rho_{\mathsf{w}_5}) \ ;^{\star} \ \overline{\overline{\{\mathsf{w}_4\}}}) \end{aligned}$$

7 Language Semantics

We refer to the variant of QWeSST$^-$ obtained by annotating the service types with the flow they advertise and extending service calls with policies as QWeSST$^\varphi$. We will now define it formally on the basis of the notions of data flow and policy introduced in Sections 4 and 5 respectively.

Although inferred flows will be most useful as part of service type annotations, it will be convenient to associate them with all free variables during typechecking. We will do so by including these annotations in their context entry. We will also need to decorate the argument of function types with a flow. As a result, QWeSST$^\varphi$ has the following syntax (base types stay unchanged):

$$\begin{array}{ll} \text{Flow types} & \tau ::= \mathsf{unit} \mid \tau \times \tau' \mid \sigma(\!|\underline{\mu}|\!) \to \tau \mid \sigma(\!|\underline{\mu}|\!) \rightsquigarrow \tau \\ \text{Expressions} & e ::= x \mid \lambda x\!:\!\tau.\, e \mid e_1\; e_2 \mid \langle e_1, e_2\rangle \mid \mathsf{fst}\; e \mid \mathsf{snd}\; e \mid () \\ & \quad \mid \mathsf{w}/u \mid \mathsf{publish}\; x : \tau.e \mid \mathsf{call}\; e_1\; \mathsf{with}\; e_2[\![\rho]\!] \mid \textit{expect}\; e\; \textit{from}\; \mathsf{w} \end{array}$$

Flow annotations will be inferred by our type system, which means that the programmer does not need to specify any flow $\underline{\mu}$ in his/her expressions. Policies ρ are instead entered by the user. Recall also that *expect* e *from* w is an internal artifact of our evaluation semantics and is also invisible to the user.

Note that our original call construct is simply a service call protected with the maximally permissive policy, i.e., $\top$. Therefore, we can define that unprotected service call introduced earlier as the derived form $\mathsf{call}\; e_1\; \mathsf{with}\; e_2 \triangleq \mathsf{call}\; e_1\; \mathsf{with}\; e_2[\![\top]\!]$.

7.1 Static Semantics

The typing semantics of QWeSST$^\varphi$ extends the typing rules of QWeSST$^-$ [14] to support: 1) statically inferring the data flow of a service and returning it as a type annotation, and 2) statically verifying that the policies attached to service calls are satisfied by the data flow of that service.

The typing judgment for this semantics has the form

$$\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \tau \qquad \text{“}e \textit{ has type } \tau \textit{ at } \mathsf{w} \textit{ w.r.t. } \Sigma \textit{ and } \Gamma\text{”}$$

where the context Γ records the type of the free variables in e and the service typing table Σ lists the types of every service available in the network. In QWeSST$^\varphi$, we shall also record the inferred flow of variables. Therefore, Γ will contain declarations of the form $x : \sigma(\!|\underline{\mu}|\!)$ and entries in Σ assume the form $\mathsf{w}/u : \sigma(\!|\underline{\mu}|\!) \rightsquigarrow \tau$. These collections are formally defined as

$$\begin{array}{l} \Gamma ::= \cdot \mid \Gamma, x : \sigma(\!|\underline{\mu}|\!) \\ \Sigma ::= \cdot \mid \Sigma, \mathsf{w}/u : \sigma(\!|\underline{\mu}|\!) \rightsquigarrow \tau \end{array}$$

We will treat them as multisets. We refer to Γ as the *local flow typing context* and to Σ as the *service typing table*.

$$\frac{}{\Sigma \mid \Gamma_{\_}, x : \sigma\llparenthesis \bullet \rrparenthesis \vdash_{\mathsf{w}} x : \sigma}\;\texttt{of\_var} \qquad \frac{}{\Sigma, \mathsf{w}'/u : \sigma\llparenthesis \underline{\mu} \rrparenthesis \leadsto \tau \mid \Gamma_{\_} \vdash_{\mathsf{w}} \mathsf{w}'/u : \sigma\llparenthesis \underline{\mu} \rrparenthesis \leadsto \tau}\;\texttt{of\_url}$$

$$\frac{}{\Sigma \mid \Gamma_{\_} \vdash_{\mathsf{w}} () : \mathsf{unit}}\;\texttt{of\_unit} \qquad \frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}} e_1 : \tau \quad \Sigma \mid \Gamma' \vdash_{\mathsf{w}} e_2 : \tau'}{\Sigma \mid (\Gamma \parallel \Gamma') \vdash_{\mathsf{w}} \langle e_1, e_2 \rangle : \tau \times \tau'}\;\texttt{of\_pair}$$

$$\frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \tau \times \tau'}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \mathsf{fst}\ e : \tau}\;\texttt{of\_fst} \qquad \frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \tau \times \tau'}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \mathsf{snd}\ e : \tau'}\;\texttt{of\_snd}$$

$$\frac{\Sigma \mid \Gamma, x : \sigma\llparenthesis \underline{\mu} \rrparenthesis \vdash_{\mathsf{w}} e : \tau}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \lambda x{:}\sigma.\, e : \sigma\llparenthesis \underline{\mu} \rrparenthesis \to \tau}\;\texttt{of\_lam} \qquad \frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}} e_1 : \sigma\llparenthesis \underline{\mu} \rrparenthesis \to \tau \quad \Sigma \mid \Gamma' \vdash_{\mathsf{w}} e_2 : \sigma}{\Sigma \mid (\Gamma \parallel (\Gamma' \,;\, \underline{\mu})) \vdash_{\mathsf{w}} e_1\ e_2 : \tau}\;\texttt{of\_app}$$

$$\frac{\Sigma \mid \Gamma, x : \sigma\llparenthesis \underline{\mu} \rrparenthesis \vdash_{\mathsf{w}} e : \tau}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \mathsf{publish}\ x : \sigma.e : \sigma\llparenthesis \mathsf{w} \succ \underline{\mu} \rrparenthesis \leadsto \tau}\;\texttt{of\_publish} \qquad \frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}'} e : \tau}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \textit{expect}\ e\ \textit{from}\ \mathsf{w}' : \tau}\;\texttt{of\_expect}$$

$$\frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}} e_1 : \sigma\llparenthesis \underline{\mu} \rrparenthesis \leadsto \tau \qquad \Sigma \mid \Gamma' \vdash_{\mathsf{w}} e_2 : \sigma \qquad \underline{\mu} \models \rho}{\Sigma \mid (\Gamma \parallel (\Gamma' \,;\, \underline{\mu})) \vdash_{\mathsf{w}} \mathsf{call}\ e_1\ \mathsf{with}\ e_2[\![\rho]\!] : \tau}\;\texttt{of\_call}$$

$$\frac{}{\cdot \vdash \cdot}\;\texttt{st\_}\cdot \qquad \frac{\Sigma \vdash \Delta \quad \Sigma \mid x : \sigma\llparenthesis \underline{\mu} \rrparenthesis \vdash_{\mathsf{w}} e : \tau}{\Sigma, \mathsf{w}/u : \sigma\llparenthesis \underline{\mu} \rrparenthesis \leadsto \tau \vdash \Delta, \mathsf{w}/u \hookrightarrow x{:}\sigma.e}\;\texttt{st\_u}$$

Fig. 2. Typing and flow system

The typing judgment $\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \tau$ is located at each host w, which is where the expression e resides. Because e is local to w, so are the variables in Γ, meaning that they are implicitly typed at w. Instead, URLs used in e may refer to other hosts, which explains why entries in Σ mention URLs. The idea of localization, both for typing and evaluation, is inspired by *Lambda 5* [7,8].

The rules defining the above judgment are given in the top part of Figure 2. Ignoring flow inference for a moment, these rules are rather standard.

Data flow for each free variable in an expression is inferred during type checking. Therefore, the flow $\underline{\mu}$ in a context entry $x : \sigma\llparenthesis \underline{\mu} \rrparenthesis$ in the conclusion of any rule is calculated on the basis of the flow of x in its premises (note that context variables themselves are instead determined from the bottom up as the derivation is built). A context entry $x : \sigma\llparenthesis \_ \rrparenthesis$ indicates that the variable x is not used in the expression being typechecked. We write $\Gamma_{\_}$ to mark all variables in context Γ in this way. In rule `of_var`, the variable x is given local flow $\bullet$ while all other variables in the context are unused. Unary rules simply propagate data flow annotations downwards. Binder rules, here `of_lam` and `of_publish`, copy the data flow annotation from the context variable in their premise to the bound variable in their conclusion.

Data flow inference for rules with two typing judgments as their premises are more interesting, as we must combine the flow annotations that come from those premises for every variable. Indeed, in these cases the expression e being typechecked consists of two subexpressions, e_1 and e_2, both of which may mention a context variable x and, crucially, may make different uses of it. As a consequence, the premises that typecheck e_1 and e_2 might have two different flow typing contexts $\mathit{\Gamma}$ and $\mathit{\Gamma}'$ that mention potentially distinct data flow for x. We must combine $\mathit{\Gamma}$ and $\mathit{\Gamma}'$ into a context with common annotations for x. How to do so depends on the expression e, and indeed rules `of_pair`, `of_app` and `of_call` take three different approaches to merging local flow typing contexts:

- In the case of `of_pair`, the computation of e_1 and e_2 are independent which means that their respective usage of x does not interfere with each other. We can simply use parallel composition to combine them. This is the gist of the notation $\mathit{\Gamma} \,\|\, \mathit{\Gamma}'$ in the conclusion of this rule.
- In the case of `of_app`, the computation of e_1 and e_2 are independent, which is similar to the case of `of_pair`. However, the function part may force a flow $\underline{\mu}$ on its argument, as indicated by the type $\sigma\llparenthesis\underline{\mu}\rrparenthesis \to \tau$. Therefore, assuming a call-by-value semantics, any value substituted for a variable y in the argument e_2 will be subject to the flow $\underline{\mu}$. So if y has flow μ_2 relative to e_2 the value returned from the function call will have flow $\mu_2 \,;\, \underline{\mu}$. The notation $\mathit{\Gamma}' \,;\, \underline{\mu}$ in the conclusion of rule `of_app` post-composes $\underline{\mu}$ to all flows in $\mathit{\Gamma}'$. Now, because y may also be subject to an independent flow μ_1 from the evaluation of the function part e_1, the overall flow of y in $e_1\ e_2$ shall be $\mu_1 \,\|\, (\mu_2 \,;\, \underline{\mu})$.
- The case of `of_call` shares similarities with `of_app`. One main difference however is that the call construct carries a policy ρ. It is in this rule that we check that the advertised flow $\underline{\mu}$ of the service being invoked satisfies the policy. This is done by the third premise of this rule, $\underline{\mu} \models \rho$. A second difference is that the value of argument e_2 will be sent to the host w' that provides this service. The change of host is recorded in the flow $\underline{\mu}$ of the input argument of this service, so that $\underline{\mu}$ must have the form $\mathsf{w}' \succ \underline{\mu}'$.

The rules in Figure 2 use two operators to combine contexts, $\mathit{\Gamma} \,;\, \underline{\mu}$ and $\mathit{\Gamma} \,\|\, \mathit{\Gamma}'$. It is easy to verify that in both cases these contexts satisfy the invariant that they contain the exact same variables with the same types (but possibly different flow annotations). They are formally defined as follows:

$$\begin{cases}
\mathit{\Gamma} \,;\, \_ & = \ \mathit{\Gamma} \\
\mathit{\Gamma} \,;\, (\mathsf{w} \succ \_) & = \ \mathit{\Gamma} \,;\, (\mathsf{w} \succ \bullet) \\
\cdot \,;\, \mu' & = \ \cdot \\
(\mathit{\Gamma}, x : \sigma\llparenthesis\_\rrparenthesis) \,;\, \mu' & = \ (\mathit{\Gamma} \,;\, \mu), x : \sigma\llparenthesis\_\rrparenthesis \\
(\mathit{\Gamma}, x : \sigma\llparenthesis\mu\rrparenthesis) \,;\, \mu' & = \ (\mathit{\Gamma} \,;\, \mu), x : \sigma\llparenthesis\mu \,;\, \mu'\rrparenthesis
\end{cases}$$

$$\begin{cases}
\cdot \,\|\, \cdot & = \ \cdot \\
(\mathit{\Gamma}, x : \sigma\llparenthesis\mu\rrparenthesis) \,\|\, (\mathit{\Gamma}', x : \sigma\llparenthesis\_\rrparenthesis) & = \ (\mathit{\Gamma} \,\|\, \mathit{\Gamma}'), x : \sigma\llparenthesis\mu\rrparenthesis \\
(\mathit{\Gamma}, x : \sigma\llparenthesis\_\rrparenthesis) \,\|\, (\mathit{\Gamma}', x : \sigma\llparenthesis\mu'\rrparenthesis) & = \ (\mathit{\Gamma} \,\|\, \mathit{\Gamma}'), x : \sigma\llparenthesis\mu'\rrparenthesis \\
(\mathit{\Gamma}, x : \sigma\llparenthesis\mu\rrparenthesis) \,\|\, (\mathit{\Gamma}', x : \sigma\llparenthesis\mu'\rrparenthesis) & = \ (\mathit{\Gamma} \,\|\, \mathit{\Gamma}'), x : \sigma\llparenthesis\mu \,\|\, \mu'\rrparenthesis
\end{cases}$$

7.2 Dynamic Semantics and Meta-Theory

The key feature of QWeSST$^\varphi$ is that data flows are inferred during type checking and policies are enforced statically too. Therefore neither plays any role at run time. In this section, we give a brief account of an execution semantics for QWeSST$^\varphi$, mostly for the purpose of proving type safety. The treatment is directly adapted from [14,15], to which we refer the reader for details.

This semantics of QWeSST is expressed by the small-step judgments:

$$\begin{array}{ll} e \ \mathsf{val} & \text{“}e \textit{ is a value}\text{”} \\ \Delta\,;\, e \ \mapsto_{\mathsf{w}} \ \Delta'\,;\, e' & \text{“}\Delta; e \textit{ steps to } \Delta'; e'\text{”} \end{array}$$

where w is the host where expression e is being evaluated, and the *service repository* Δ is defined as follows:

$$\text{Global service repository} \quad \Delta ::= \cdot \ \mid \ \Delta, \mathsf{w}/u \hookrightarrow x : \sigma.e$$

Each item $\mathsf{w/u} \hookrightarrow x : \sigma.e$ is a service with URL w/u, formal argument x and body e. The repository Δ is global as it lists every service in the network. See [15] for a more realistic approach.

Data flow types and policies do not appear in the dynamic semantics. This is because our approach focuses on verifying data flow compliance statically. When an expression typechecks, it means that all service call policies are satisfied and the execution can take place.

The rules defining the above judgments can be found in [14,15]. Most are unsurprising and we only report some of the more interesting rules pertaining to remote invocations.

$$\frac{}{\Delta\,;\, \mathsf{publish}\ x : \sigma.e \ \mapsto_{\mathsf{w}} \ (\Delta, \mathsf{w}/u \hookrightarrow x : \sigma.e)\,;\, \mathsf{w}/u}\ \texttt{ev\_publish}$$

$$\frac{v_2 \ \mathsf{val}}{\underbrace{(\Delta^*, \mathsf{w}'/u \hookrightarrow x : \sigma.e)}_{\Delta}\,;\, \mathsf{call}\ \mathsf{w}'/u\ \mathsf{with}\ v_2 \ \mapsto_{\mathsf{w}} \ \Delta\,;\, \textit{expect}\ [v_2/x]\ e\ \textit{from}\ \mathsf{w}'}\ \texttt{ev\_call}_3$$

$$\frac{\Delta\,;\, e \ \mapsto_{\mathsf{w}'} \ \Delta'\,;\, e'}{\Delta\,;\, \textit{expect}\ e\ \textit{from}\ \mathsf{w}' \ \mapsto_{\mathsf{w}} \ \Delta'\,;\, \textit{expect}\ e'\ \textit{from}\ \mathsf{w}'}\ \texttt{ev\_expect}_1$$

$$\frac{v \ \mathsf{val}}{\Delta\,;\, \textit{expect}\ v\ \textit{from}\ \mathsf{w}' \ \mapsto_{\mathsf{w}} \ \Delta\,;\, v}\ \texttt{ev\_expect}_2$$

The evaluation of $\mathsf{publish}\ x : \tau.e$ immediately publishes its argument as a web service in the repository, creating a new unique identifier for it and returning the corresponding URL. To call a web service, we first reduce its first argument to a URL, its second argument to a value, and then carry out the remote invocation which is modeled using the internal construct $\textit{expect}\ [v_2/x]e\ \textit{from}\ \mathsf{w}'$. This implements the client's inactivity while awaiting for the server w' to evaluate $[v_2/x]e$ to a value. This is done in rules $\texttt{ev\_expect}_1$ and $\texttt{ev\_expect}_2$: the former performs one step of computation on the server w' while the client w is essentially waiting. Once this expression has been fully evaluated, the latter rule kicks in and delivers the result to the client. A more realistic multi-threaded semantics can be found in [15].

The bottom part of Figure 2 defines the judgment $\Sigma \vdash \Delta$ which specifies that service repository Δ is well-typed with respect to typing table Σ. This judgment is used to prove that QWeSST$^{\varphi}$ is type-safe.

Like QWeSST, QWeSST$^{\varphi}$ admits localized versions of type preservation and progress, thereby making it a type safe language. The techniques used to prove these results are fairly traditional. We used the Twelf proof assistant [11] to encode each of our proofs and to verify their correctness. These proofs are very similar to those for QWeSST, which can be found in [15].

Theorem 1 (Type preservation). *If $\Delta\,;\,e \;\mapsto_{\mathsf{w}}\; \Delta'\,;\,e'$ and $\Sigma \mid \cdot \vdash_{\mathsf{w}} e : \tau$ and $\Sigma \vdash \Delta$, then $\Sigma' \mid \cdot \vdash_{\mathsf{w}} e' : \tau$ and $\Sigma' \vdash \Delta'$.*

Theorem 2 (Progress). *If $\Sigma \mid \cdot \vdash_{\mathsf{w}} e : \tau$ and $\Sigma \vdash \Delta$, then*
- *either e val,*
- *or there exist e' and Δ' such that $\Delta\,;\,e \;\mapsto_{\mathsf{w}}\; \Delta'\,;\,e'$.*

8 Related Work

In this work, we examined the dependencies between third-party web services, focusing on data flow and policies to control them. This is related to the problem of analyzing library dependencies of non-distributed and single-threaded programs (also called data flow graph) [12]. This was adapted to the context of web services in [2,3], which proposes a model for combining web services. The difference between these approaches and our work is that, in QWeSST$^{\varphi}$, services can be created dynamically as opposed to static library or service identifiers. This is the reason why we introduced specific flow types to be able to infer the data flow of a program statically. This idea is also found in the history-based type systems proposed in [1].

Verifying policy constraints statically entails some restrictions. For instance, we cannot express a policy that depends on the value of an expression. However, if the program is correctly typed, it guarantees that the policies are satisfied and the program can be executed. Similar ideas were explored in [16,17].

The development of QWeSST$^{\varphi}$ shares concerns with recent work from Collinson and Pym. In [4,5], they propose to use bunched logic to specify acceptable composition of shared resources. The work outlined here is specific to the context of web programming.

Our use of the term *data flow* should not be confused with *information flow* and the large body of work on related concepts such as non-interference and declassification (see for example [9,13]). The problems they address (inferring and controlling where data will go in a distributed program and preventing information leakage in shared multi-user systems) are quite different, possibly orthogonal. Yet, it will be interesting to see how these two approaches can be used in conjunction to provide a robust security model for a distributed programming language.

The formalization of policies aimed at preventing unwanted dissemination of possibly sensitive data is a theme that this work has in common with P3P [6] and related approaches to privacy. QWeSST$^\varphi$ is not specifically designed for privacy applications, although it can express some simple privacy policies. On the other hand, privacy-oriented systems such as P3P do not try to infer the flow of data within a network, but to establish and verify precise agreements between the consumer and the provider of a service.

9 Conclusions and Future Work

In this paper, we introduced a proof of concept for a web based programming language that automatically predicts how third-party services are composed and enables service consumers to specify policies that control data sent as inputs to these third-party services. We started from a fragment of QWeSST, a simple programming language, and extended it in two ways: first, we defined a data flow model that allows us to describe the paths taken by a value through a network of web services. Second, we defined a policy language to express constraints on acceptable paths that a value should take. This allowed us to automatically infer an approximation of the data flows of program variables and verify that the policies attached to service calls are satisfied. This verification is done statically. Therefore, since typechecking takes place locally in the resulting language, QWeSST$^\varphi$, it means that the service consumer has the guarantee that if one of its policy was violated the language interpreter would not allow the execution to take place, thereby preventing any data leakage.

QWeSST$^\varphi$, as presented in this paper, has several limitations that we intend to resolve in the near future. A first limitation is that functions and services must have arguments of base type, thereby preventing the definition of higher-order functions and services. We are investigating ways to allow such entities, which currently seem to require significant changes to our language. A second limitation is is that, in QWeSST$^\varphi$, flows and policies refer to nodes, which prevents discriminating between two services, possibly one trusted and the other one not, coming from the same service provider.

In future work, we also want to refine our data flow model to incorporate a form of polymorphism at the level of hosts. By doing so, we still would be able to verify data flow policies statically since the interpreter would be able to infer potential hosts and potential URLs involved in an expression. Another avenue of future work is concerned with the direction of our flow inference: in QWeSST$^\varphi$, we infer where a value supplied to a service may go. Another interesting problem is to try to infer, statically, where data is coming from.

Acknowledgments

This work was partially supported by the Qatar National Research Fund under grants JSREP 1-033-1-006 and NPRP 09-1107-1-168.

References

1. Martín Abadi and Cédric Fournet. Access control based on execution history. In The Internet Society, editor, *Network and Distributed System Security Symposium, NDSS*, San Diego, CA, 2003.
2. Massimo Bartoletti, Pierpaolo Degano, Gian Ferrari, and Roberto Zunino. Model checking usage policies. *Trustworthy Global Computing*, pages 19–35, 2009.
3. Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. History-based access control with local policies. *Foundations of Software Science and Computational Structures*, pages 316–332, 2005.
4. Matthew Collinson and David J. Pym. Algebra and logic for resource-based systems modelling. *Mathematical Structures in Computer Science*, 19(5), 2009.
5. Matthew Collinson and David J. Pym. Algebra and logic for access control. *Formal Aspects of Computing*, 22(2), 2010.
6. Lorrie Faith Cranor and Joseph Reagle. The platform for privacy preferences. *Communications of the ACM*, 42(2):48–55, 1999.
7. T. Murphy VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon University, January 2008. Available as technical report CMU-CS-08-126.
8. Tom Murphy VII, Karl Crary, and Robert Harper. Type-Safe Distributed Programming with ML5. *Trustworthy Global Computing*, pages 108–123, 2008.
9. Andrew C. Myers. JFlow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM.
10. Siani Pearson. *Trusted Computing Platforms: TCPA Technology In Context*. Prentice Hall PTR, July 2002.
11. Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th Int. Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, 1999. Springer-Verlag LNAI 1632.
12. NY Jeanne Ferrante Ron Cytron IBM Research Division, Yorktown Heights, Barry K. Rosen, N. Wegman Mark, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1991.
13. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, Jan 2003.
14. Thierry Sans and Iliano Cervesato. QWeSST for Type-Safe Web Programming. In Berndt Farwer, editor, *Third International Workshop on Logics, Agents, and Mobility — LAM'10*, Edinburgh, Scotland, UK, 2010.
15. Thierry Sans and Iliano Cervesato. Type-Safe Web Programming in QWeSST. Technical Report CMU-CS-10-125, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 2010.
16. Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. *Security and Privacy, IEEE Symposium on*, 0:369–383, 2008.
17. Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2):67–84, 2007.