Security issues in service composition

Massimo Bartoletti Pierpaolo Degano Gian Luigi Ferrari

Dipartimento di Informatica, Università di Pisa, Italy
{bartolet, degano, giangi}@di.unipi.it

Abstract. We use a distributed, enriched λ-calculus for describing networks of services. Both services and their clients can protect themselves, by imposing security constraints on each other's behaviour. Then, service interaction results in a call-by-property mechanism, that matches the client requests with service's. A static approach is also described, that determines how to compose services while guaranteeing that their execution is always secure, without resorting to any dynamic check.

1 Introduction

Service-oriented computing (SOC) is an emerging paradigm to design distributed applications [31, 30, 19]. In this paradigm, applications are built by assembling together independent computational units, called *services*. A service is a standalone component distributed over a network, and made available through standard interaction mechanisms. An important aspect is that services are *open*, in that they are built with little or no knowledge about their operating environment, their clients, and further services therein invoked. Composition of services may require peculiar mechanisms to handle complex interaction patterns (e.g. to implement transactions), while enforcing non-functional requirements on the system behaviour (e.g. security and service level agreement). Web Services [3, 34, 38] built upon XML technologies are possibly the most illustrative and well developed example of the SOC paradigm. Indeed, a variety of XML-based technologies already exists for describing, discovering and invoking web services [18, 14, 5, 39]. There are also several standards for defining and enforcing non-functional requirements of services, e.g. WS-Security [6], WS-Trust [4] and WS-Policy [15] among the others.

Security and service composition

The *orchestration* of services consists of their composition and coordination. Languages for that have been recently proposed, e.g. BPEL4WS [5, 25]. Service composition heavily depends on which information about a service is made public, on how to choose those services that match the user's requirements, and on their actual run-time behaviour. Security makes service composition even harder. Services may be offered by different providers, which only partially trust each other. On the one hand, providers have to guarantee the delivered service to respect a given security policy, in any interaction with the operational environment, and regardless of who actually called the service. On the other hand, clients may want to protect their sensible data from the services invoked.

A typical approach consists in endowing the network infrastructure with authentication mechanisms, so to certify the identity of services. However, security may be breached even by trusted services, either because of unintentional behaviour (e.g. bugs), or because the composition of the client and the services exhibits some behaviour unwanted by the client (e.g. leakage of information).

We have addressed the problem of security in a linguistic framework. In our approach, clients may protect from their callers by wrapping security-critical portions of their own code into *safety framings*. These framings enforce the given security policy on the execution of the wrapped piece of code, aborting it whenever about to violate the policy, thus offering additional flexibility with respect to monolithic *global* policies, and relieving the programmer of guarding each use of security-critical resources.

On their side, callers may constrain the behaviour of the called services, by supplying a security policy at the moment of invocation. We push further this invocation mechanism, by allowing callers to request services that not only do obey the imposed security constraints, but that also respect a given contract on their functional behaviour. The implementation of this so-called *call-by-property* invocation mechanism requires that services are published together with a *certified* abstraction of their behaviour.

The planning problem

Call-by-property invocation and safety framings make service composition secure. A *plan* orchestrates the execution of a service-based application, by associating the sequence of run-time service requests with a corresponding sequence of selected services. A major problem is still left open: how to construct a plan that guarantees no executions will abort because of some action attempting to violate security.

Determining such a viable plan amounts to selecting from the network those services that accomplish the requested task, while respecting the security constraints on demand. Those services that locally obey the property imposed by a request are not always good candidates, because their behaviour may affect security of the whole composition. For example, consider a device with a limited computational power. Suppose it downloads an applet from the network, and then delegates a remote service to run it. Although the contract between the device and the code provider is fulfilled, the applet may violate a security policy imposed by the executer. To determine the viable plans, one has to check the effects of all the available applets against the security policies of all the remote executers.

As a matter of fact, there might be several different kinds of plans, each with a different expressive power. Among them, one may consider plans that attach a selection of services to each program point representing a service request. The expressive power varies according to the nature of the information associated with each request. *Simple plans* associate a single service with each request, *multi-choice plans* map requests into sets of services, and *dependent plans* also convey the dependence of a service selection with the choices made in the past (a sort of continuation-passing plan). These kinds of plans have been studied in [9]. *Dependent multi-choice* plans are a mix of the last two kinds. Further expressive power is gained when relaxing the assumption of associating service selections to the program points where requests are made. *Regular plans* drive the execution of a program, by providing it with the possible patterns of service selections, in the form of a regular expression. *Dynamic plans* can be updated at run-time, according to the evaluation of some conditions on the program execution (e.g. boolean guards in conditionals, number of iterations in a loop, etc.).

A static approach to secure service composition

We have proposed a solution to the planning problem, within a distributed framework [10]. Services are functional units in an enriched λ-calculus, they are explicitly located at network sites, and they have a published public interface. Unlike standard syntactic signatures, this interface includes an abstraction of the service behaviour, in the form of annotated types. To obtain a service with a specific behaviour, a client queries the network for a published interface matching the requirements. Security is implemented by wrapping the critical blocks of code inside safety framings (with local scopes, possibly nested), that enforce the relevant policies during the execution of the block. In the spirit of history-based security [1], a security policy can inspect the whole execution history at a given site. Since our framework is fully distributed, our policies cannot span over multiple sites.

We have introduced a type and effect system for our calculus [21, 28, 35]. The type of a service describes its functional behaviour, while the effect is a *history expression*, representing those histories of events relevant to security. History expressions extend regular expressions with information about the selection of services, coupled with their corresponding effect.

We have then devised a way of extracting from a history expression all the *viable* plans, i.e. those that successfully drive secure executions. This is a two-stage construction. A first transformation of history expressions makes them model-checkable for validity [7]. Valid history expressions guarantee that the services they come from never go wrong at run-time. From valid histories it is then immediate to obtain the viable plans, that make any execution monitor unneeded.

Trusted orchestration

Our planning technique acts as a *trusted orchestrator* of services. It provides a client with the plans guaranteeing that the invoked services always respect the required properties. Thus, in our framework the only trusted entity is the orchestrator, and neither clients nor services need to be such. In particular, the orchestrator infers functional and behavioural types of each service. Also, it is responsible for certifying the service code, for publishing its interface, and for guaranteeing that services will not arbitrarily change their code on the fly: when this happens, services need to be certified again.

When an application is injected in the network, the orchestrator provides it with a viable plan (if any), constructed by composing and analysing the certified interfaces of the available services. The trustworthiness of the orchestrator relies upon formal grounds. We proved the soundness of our type and effect system, and the correctness of the static analysis and model-checking technique that infers viable plans.

The orchestrator constructs the plans for a client, by considering the view of the network at the moment the application is injected. To be more dynamic, one would like to manage the discovering of new services, as well as the case when existing ones are no longer available.

Both these problems require a special treatment. Multi-choice plans are a first solution to deal with disappearing services, because they offer many choices for the same request. Publication of new services poses instead a major problem. To cope with that, one has to reconfigure plans at run-time, by exploiting the new interfaces. However, incrementally checking viability of plans is an open problem. A possible solution is to enrich history expressions with *hooks* where new services can be attached. The orchestrator then needs to check the validity of the newly discovered plans, hopefully in an incremental manner.

Related work

The secure composition of components underlies the design of Sewell and Vitek's box-π [33], an extension of the π-calculus that allows for expressing safety policies in the form of *security wrappers*. These are programs that encapsulate a component to control the interactions with other (possibly untrusted) components. A type system that statically captures the allowed causal information flows between components. Our safety framings are closely related to wrappers.

Gorla, Hennessy and Sassone [23] consider a calculus for agents which may migrate between sites in a controlled manner. Each site has a *membrane*, representing both a security policy and a classification of the levels of trust of external sites. A membrane guards the incoming agents before allowing them to execute.

Recently, increasing attention has been devoted to express service contracts as behavioural (or session) types. These synthetise the essential aspects of the interaction behaviour of services, while allowing efficient static verification of properties of composed systems. Session types [24] have been exploited to formalize compatibility of components [37] and to describe adaptation of web services [16]. Security issues have been recently considered in terms of session types, e.g. in [13], which proves the decidability of type-checking in an extension of the π-calculus with session types and correspondence assertions [40].

Other works have proposed type-based methodologies to check security properties of distributed systems. For instance, Gordon and Jeffrey [22] use a type and effect system to prove authenticity properties of security protocols. Web service authentication has been recently modelled and analysed in [11, 12] through a process calculus enriched with cryptographic primitives.

The problem of discovering and composing Web Services by taking advantage of semantic information has been the subject of a considerable amount of research and development, [2, 17, 27, 29, 32, 36] to cite a few. The idea is to extend the primitives of service description languages with basic constructs for specifying properties of the published interface. We can distinguish between semantic-web descriptions [2, 29, 32, 36] in which service interfaces are annotated with parameter ontologies, and behavioural description [17, 27] in which the annotation details the ordering of service actions. A different solution to planning service composition has been proposed in [26], where the problem of composing services in order to achieve a given goal is expressed as a constraint satisfaction problem. Our approach extends and complements those based on behavioral descriptions, with an eye to security. Indeed, our methodology fully automates the process of discovering services and planning their composition in a secure way.

2 Planning secure service compositions

To illustrate our approach, consider the scenario in the figure below. The boxes model services, distributed over a network. Each box encloses the service code, and is decorated with the location ℓ_i where the service is published.

Assume that the client at site ℓ_0 is a device with limited computational capabilities, wanting to execute some code downloaded from the network. To do that, the client issues two requests in sequence. The request labelled r_1 asks for a piece of mobile code (e.g. an applet), and it can be served by two code providers at ℓ_1 and ℓ_2. The request type $\tau \to (\tau \to \tau)$ means that, upon receiving a value

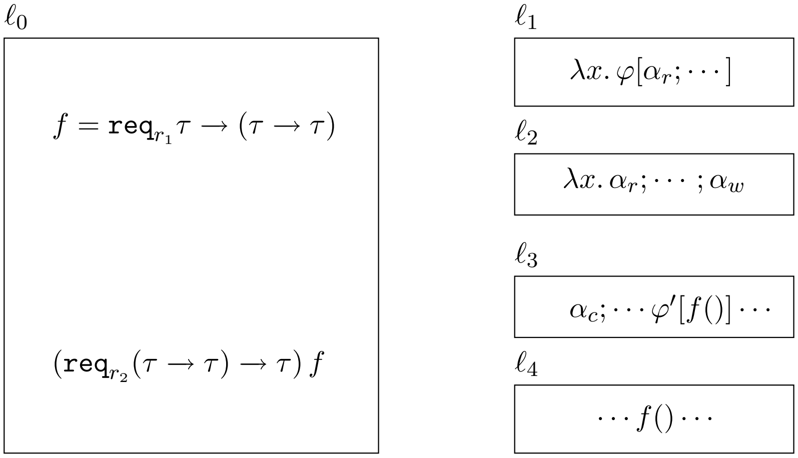

ℓ_0
$f = \mathtt{req}_{r_1} \tau \to (\tau \to \tau)$
$(\mathtt{req}_{r_2} (\tau \to \tau) \to \tau)\, f$
ℓ_1
$\lambda x.\, \varphi[\alpha_r; \cdots]$
ℓ_2
$\lambda x.\, \alpha_r; \cdots ; \alpha_w$
ℓ_3
$\alpha_c; \cdots \varphi'[f()] \cdots$
ℓ_4
$\cdots f() \cdots$

Fig. 1. One client (ℓ_0), two code providers (ℓ_1, ℓ_2), and two code executers (ℓ_3, ℓ_4).

of type τ (which can be an arbitrary base type, immaterial here) the invoked service replies with a function from τ to τ, with no security constraints.

The service at ℓ_1 returns a function that protects itself with a policy φ, permitting its use in certified sites only (modelled by the event α_c). Within the function body, the only security-relevant operation is a read α_r on the file system where the delivered code is run. The code provided by ℓ_2 first reads (α_r) some local data, and eventually writes them (α_w) back to ℓ_2.

Since ℓ_0 has a limited computational power, the code f obtained by the request r_1 is passed as a parameter to the service invoked by the request r_2. This request can be served by ℓ_3 and ℓ_4. The service at ℓ_3 is certified (α_c), and runs the provided code f under a “Chinese Wall” security policy φ', requiring that no data can be written (α_w) after reading them (α_r). The service at ℓ_4 is not certified, and it simply runs f.

Programming model

Clients and services are modelled as expressions in a λ-calculus enriched with primitives for security and service requests. Security-relevant operations are rendered as side-effects in the calculus, and they are called *access events* (e.g. $\alpha_c, \alpha_r, \alpha_w$). A *security policy* is a regular property over a sequence η of access events, namely a *history*. A piece of code e framed within a policy φ (written $\varphi[e]$) must respect φ at each step of its execution. A *service request* has the form $\mathtt{req}_r \rho$. The label r uniquely identifies the request, while the *request type* ρ is the query pattern to be matched by the invoked service. For instance, the request type $\tau \xrightarrow{\varphi[\bullet]} \tau'$ matches services with functional type $\tau \to \tau'$, and whose behaviour respects the policy φ. The abstract syntax of services follows.

Syntax of services

e, e'	$::=$	x	variable
		α	access event
		$\mathtt{if}\, b\, \mathtt{then}\, e\, \mathtt{else}\, e$	conditional
		$\lambda_z x.\, e$	named abstraction
		$e\, e'$	application
		$\varphi[e]$	safety framing
		$\mathtt{req}_r \rho$	service request
		$\mathtt{wait}\, \ell$	wait reply

The stand-alone evaluation of a service is much alike the call-by-value semantics of the λ-calculus; additionally, it enforces all the policies within their framings. More precisely, assume that, starting from the current history η, an expression e may evolve to e' and extend the history to η'. Then, a framing $\varphi[e]$ may evolve to $\varphi[e']$ if η' satisfies φ — otherwise the evaluation gets stuck. Eventually, values leave the scope of framings.

When a service is plugged into a network, a plan is used to resolve the requests therein, acting as an orchestrator. For brevity, we consider here only the case of *simple* plans, that have the following syntax:

Syntax of Simple Plans

$$\pi, \pi' \quad ::= \quad 0 \quad | \quad r[\ell] \quad | \quad \pi \mid \pi'$$

The empty plan 0 has no choices. The plan $r[\ell]$ associates the service e published at site ℓ with the request labelled r. The composition operator $|$ is associative, commutative and idempotent, with identity 0. We require plans to have a single choice for each request, i.e. $r[\ell] \mid r[\ell']$ implies $\ell = \ell'$.

A service e is plugged into a network by publishing it at a site ℓ, together with its interface τ. Hereafter, $\ell\langle e : \tau\rangle$ denotes such a *published service*. We assume that each site publishes a single service, and that interfaces are certified, e.g. they are inferred by the type system in [10]. Also, we assume that services cannot invoke each other circularly, because this would make little sense. A *network* is a set of clients and published services.

The state of a published service $\ell\langle e : \tau\rangle$ is denoted by $\ell\langle e : \tau\rangle : \eta, e'$ — abbreviated as $\ell : \eta, e'$ when unambiguous. The component η is the history generated so far at site ℓ, and e' models the code in execution. We assume here that services are *stateless*, i.e. the history of a service is cleared at each instantiation. A network configuration has the form $\ell_1 : \eta_1, e'_1 \parallel \cdots \parallel \ell_k : \eta_k, e'_k$.

A request r, resolved by the current plan with the service ℓ', can be served if the service ℓ' is available (written $\ell' : \star$). In this case, a new instance of the service is generated: the service code (a function) is applied to the received argument. The invoker waits until ℓ' has produced a value. When this happens, the value is returned to the invoker, and the service becomes available again.

Back to our example, consider the plan $\pi = r_1[\ell_2] \mid r_2[\ell_3]$. Then, π drives the following computation (for brevity, we omit the types in requests):

$$\begin{array}{l}
\quad \ell_0 : \varepsilon, (\lambda f.\, \mathtt{req}_{r_2} f)\ \mathtt{req}_{r_1} \parallel \ell_1 : \star \parallel \ell_2 : \star \parallel \ell_3 : \star \parallel \ell_4 : \star \\
\to_\pi \ell_0 : \varepsilon, (\lambda f.\, \mathtt{req}_{r_2} f)\ \mathtt{wait}\, \ell_2 \parallel \ell_1 : \star \parallel \ell_2 : \varepsilon, \lambda x.\, \alpha_r; \cdots ; \alpha_w \parallel \ell_3 : \star \parallel \ell_4 : \star \\
\to_\pi \ell_0 : \varepsilon, (\lambda f.\, \mathtt{req}_{r_2} f)\, (\lambda x.\, \alpha_r; \cdots ; \alpha_w) \parallel \ell_1 : \star \parallel \ell_2 : \star \parallel \ell_3 : \star \parallel \ell_4 : \star \\
\to_\pi \ell_0 : \varepsilon, \mathtt{req}_{r_2} (\lambda x.\, \alpha_r; \cdots ; \alpha_w) \parallel \ell_1 : \star \parallel \ell_2 : \star \parallel \ell_3 : \star \parallel \ell_4 : \star \\
\to_\pi \ell_0 : \varepsilon, \mathtt{wait}\, \ell_3 \parallel \ell_1 : \star \parallel \ell_2 : \star \parallel \ell_3 : \varepsilon,\, \alpha_c; \cdots ; \varphi'[(\lambda x.\, \alpha_r; \cdots ; \alpha_w)()] \parallel \ell_4 : \star \\
\to_\pi \ell_0 : \varepsilon, \mathtt{wait}\, \ell_3 \parallel \ell_1 : \star \parallel \ell_2 : \star \parallel \ell_3 : \alpha_c,\, \varphi'[(\lambda x.\, \alpha_r; \cdots ; \alpha_w)()] \parallel \ell_4 : \star \\
\to_\pi \ell_0 : \varepsilon, \mathtt{wait}\, \ell_3 \parallel \ell_1 : \star \parallel \ell_2 : \star \parallel \ell_3 : \alpha_c,\, \varphi'[\alpha_r; \cdots ; \alpha_w] \parallel \ell_4 : \star \\
\to_\pi \ell_0 : \varepsilon, \mathtt{wait}\, \ell_3 \parallel \ell_1 : \star \parallel \ell_2 : \star \parallel \ell_3 : \alpha_c \alpha_r,\, \varphi'[\alpha_w] \parallel \ell_4 : \star
\end{array}$$

The computation at site ℓ_3 is now aborted, because the history $\alpha_c\alpha_r\alpha_w$ would otherwise violate the Chinese-Wall policy φ'. We have then discovered that the plan $r_1[\ell_2] \mid r_2[\ell_3]$ is *not* viable. As we will see in a while, our static machinery infers that also the plan $r_1[\ell_1] \mid r_2[\ell_4]$ is not viable (it violates the policy φ). There are two further plans to consider: $r_1[\ell_1] \mid r_2[\ell_3]$ and $r_1[\ell_2] \mid r_2[\ell_4]$. These plans will be shown viable by our static analysis, and they will drive secure executions that never abort.

Types and effects

We stipulated that the services published in the network have certified interfaces. To do that, in [10] we have defined a type and effect system, that will also be used to infer an over-approximation of client behaviour.

Types τ, τ' are either base types, or they have the form $\tau \xrightarrow{H} \tau'$. The annotation H over the arrow is a *history expression.* It describes the latent effect associated with the function: one of the histories represented by H is generated when a value is applied to a function with that type. History expressions have the following abstract syntax:

Syntax of history expressions

H, H'	$::=$	ε	empty
		h	variable
		α	access event
		$H \cdot H'$	sequence
		$H + H'$	choice
		$\varphi[H]$	safety framing
		$\mu h.H$	recursion
		$\ell : H$	localization
		$\{\pi_i \rhd H_i\}_{i \in I}$	planned selection

Access events represent the program actions where sensible resources are accessed; the constructors $\cdot$ and $+$ correspond to sequentialization of code and conditionals, respectively; safety framings model blocks of code subject to security policies; recursion is for loops and recursive functions. The construct $\ell : H$ localizes the behaviour H to the site ℓ. E.g., $\ell : \alpha \cdot (\ell' : \alpha') \cdot \beta$ denotes two histories: $\alpha\beta$ occurring at location ℓ, and α' occurring at ℓ'. A planned selection abstracts the behaviour of service requests. E.g., $\{r[\ell_1] \rhd H_1 \cdots r[\ell_k] \rhd H_k\}$ says that a request r can be resolved into one of the services provided by the sites $\ell_1, \ldots, \ell_k$, which may generate a history represented by $H_1, \ldots, H_k$, respectively.

A typing judgment $\Gamma, H \vdash_\ell e : \tau$ means that the service e at site ℓ evaluates to a value of type τ, and produces a history denoted by the effect H. Typing judgments are similar to those of the simply-typed λ-calculus. To give the flavour of how the effects are inferred, consider first the rule to type applications:

$$\frac{\Gamma, H \vdash_\ell e : \tau \xrightarrow{H''} \tau' \quad \Gamma, H' \vdash_\ell e' : \tau}{\Gamma, H \cdot H' \cdot H'' \vdash_\ell e\, e' : \tau'}$$

The rule says that e is an expression whose evaluation will generate a history in H. It will reduce to a value which is a function (from τ to τ') with latent effect H''. The evaluation of the argument e' with type τ will generate a history in H'. The overall effect of e applied to e' is $H \cdot H' \cdot H''$, thus respecting the evaluation order of the call-by-value semantics (function, argument, latent effect).

To give a type to service requests, some auxiliary technical notation is needed: the interested reader can find all the definitions in [7]. Just to give the intuition, the typing judgement for a request $\mathtt{req}_r\rho$ has the following schema:

$$\frac{\tau = \Cup\{\, \rho \boxplus_{r[\ell']} \tau' \mid \emptyset, \varepsilon \vdash_{\ell'} e : \tau' \quad \ell \prec \ell'\langle e : \tau'\rangle \quad \rho \approx \tau' \,\}}{\Gamma, \varepsilon \vdash_{\ell} \mathtt{req}_r\rho : \tau}$$

Requests have an empty actual effect, and a functional type τ. The latent effect is a planned selection that picks from the network those services known by ℓ and matching the request type ρ. The relation $\rho \approx \tau$ models ρ being compatible with τ; the partial order $\prec$ represents visibility among services (e.g. $\ell \prec \ell'$ when ℓ' is visible by ℓ). The operator $\boxplus$ combines the request type with a service interface; the operator $\Cup$ suitably assembles such combinations into a planned selection.

Back to our running example, the types inferred for the services are shown in the figure below (they are displayed as decorations of the boxes). The public interface of the client is the base type *unit*, meaning that it cannot be invoked by other services. The interfaces of the services are as expected; for instance, the type of ℓ_3 is a function that, when applied to a function with latent effect h, will produce a value of type τ, and a history denoted by $\alpha_c \cdot \varphi'[h]$.

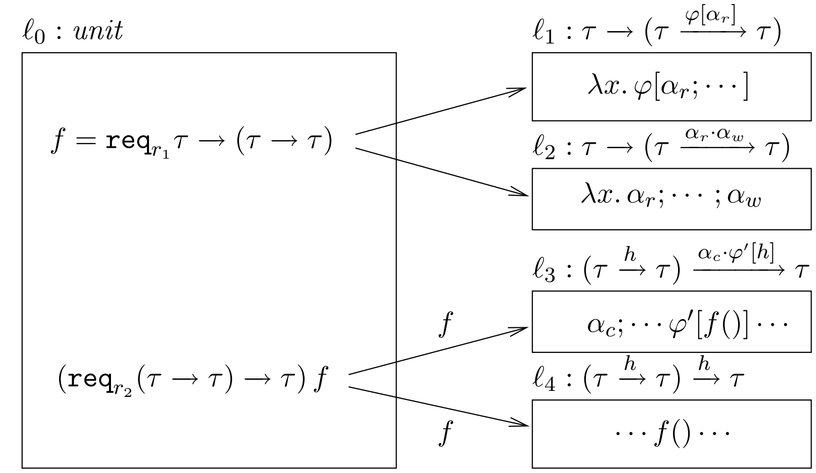

$\ell_0 : unit$
$f = \mathtt{req}_{r_1} \tau \to (\tau \to \tau)$
$(\mathtt{req}_{r_2} (\tau \to \tau) \to \tau)\, f$
$\ell_1 : \tau \to (\tau \xrightarrow{\varphi[\alpha_r]} \tau)$
$\lambda x.\, \varphi[\alpha_r; \cdots]$
$\ell_2 : \tau \to (\tau \xrightarrow{\alpha_r \cdot \alpha_w} \tau)$
$\lambda x.\, \alpha_r; \cdots ; \alpha_w$
$\ell_3 : (\tau \xrightarrow{h} \tau) \xrightarrow{\alpha_c \cdot \varphi'[h]} \tau$
f
$\alpha_c; \cdots \varphi'[f()] \cdots$
$\ell_4 : (\tau \xrightarrow{h} \tau) \xrightarrow{h} \tau$
f
$\cdots f() \cdots$

Fig. 2. One client, four services, and their (certified) published interfaces.

To obtain an over-approximation to the behaviour of the network upon the injection of a client, our type and effect system suitably combines the abstract behaviour of the client with the certified interfaces of the services it can invoke.

The inferred history expression approximates the run-time behaviour of *each* site in the network. For our running example, the abstract behaviour of the whole network is rendered by the following history expression H:

$$\begin{aligned}\{ & r_2[\ell_3] \rhd \ell_3 : \alpha_c \cdot \varphi'[\{r_1[\ell_1] \rhd \varphi[\alpha_r], r_1[\ell_2] \rhd \alpha_r \cdot \alpha_w\}] \\ & r_2[\ell_4] \rhd \ell_4 : \{r_1[\ell_1] \rhd \varphi[\alpha_r], r_1[\ell_2] \rhd \alpha_r \cdot \alpha_w\}\}\end{aligned}$$

The intuitive meaning of H is that, under the plan $r_2[\ell_3]$ — i.e. if r_2 is served by ℓ_3 — the event α_c is generated at site ℓ_3, followed by a safety framing φ'. This framing wraps $\varphi[\alpha_r]$ if ℓ_1 is chosen for r_1, or $\alpha_r\alpha_w$ if ℓ_2 is chosen instead. Otherwise, if r_2 is served by ℓ_4, then the behaviour (on site ℓ_4) depends on the former choice for r_1: if ℓ_1 was selected, then $\varphi[\alpha_r]$, otherwise $\alpha_r\alpha_w$. Note also that no event is generated by the client at site ℓ_0.

Say that a computation *goes wrong at* ℓ when it reaches a configuration whose state at ℓ is stuck. For example, a configuration $\eta, \varphi[e]$ is stuck if a step of e would violate φ. Say a plan π *viable* for e at ℓ when the evolution of e within a network, under plan π, does not go wrong at ℓ. Then, we say that a history expression H is π-valid when the plan π is viable for all the histories produced by H under π.

For example, consider the history expression $H_1 = \alpha_c \cdot \varphi'[\alpha_r \cdot \alpha_w]$, where φ' requires that no write α_w occurs after a read α_r. Then, H_1 is *not* 0-valid. Indeed, under the empty plan 0, the event α_w occurs within a safety framing enforcing φ', and the history $\alpha_c\alpha_r\alpha_w$ does not obey φ'.

We have proved two fundamental results about our type and effect system. First, it correctly over-approximates the actual run-time histories. Second, it enjoys the following type safety property.

Theorem 1. *Let* $\{\ell_i\langle e_i : \tau_i\rangle\}_{i \in I}$ *be a network, and let* $\emptyset, H_i \vdash e_i : \tau_i$ *for all* $i \in I$. *If* H_i *is* π_i*-valid, then* π_i *is viable for* e_i *at* ℓ_i.

Therefore, to find the viable plans for a client, one has to infer the effect H of the client injected in the network, and then find the plans π_i that make H π_i-valid. The following sections show how to do that.

Extracting viable plans I: Linearizing history expressions

Once extracted a history expression H from a client e, we analyse H to find if there is any viable plan for the execution of e. This issue is not trivial, because the effect of selecting a given service for a request is not confined to the execution of that service. Since each service selection affects the *whole* execution of a network, we cannot simply devise a viable plan by selecting services that satisfy the constraints imposed by the requests, only.

This is actually shown by our running example. Consider again the aborting computation with plan $r_1[\ell_2] \mid r_2[\ell_3]$, here slightly abridged:

$$\begin{aligned}
& \ell_0 : \varepsilon, (\lambda f.\, \texttt{req}_{r_2} f)\ \texttt{req}_{r_1} \parallel \ell_2 : \star \parallel \ell_3 : \star \\
\to_\pi\ & \ell_0 : \varepsilon, (\lambda f.\, \texttt{req}_{r_2} f)\ \texttt{wait}\, \ell_2 \parallel \ell_2 : \varepsilon, \lambda x.\, \alpha_r; \cdots ; \alpha_w \parallel \ell_3 : \star \\
\to_\pi^*\ & \ell_0 : \varepsilon, \texttt{req}_{r_2} (\lambda x.\, \alpha_r; \cdots ; \alpha_w) \parallel \ell_2 : \star \parallel \ell_3 : \star \\
\to_\pi\ & \ell_0 : \varepsilon, \texttt{wait}\, \ell_3 \parallel \ell_2 : \star \parallel \ell_3 : \varepsilon,\ \alpha_c; \cdots ; \varphi'[(\lambda x.\, \alpha_r; \cdots ; \alpha_w)()] \\
\to_\pi^*\ & \ell_0 : \varepsilon, \texttt{wait}\, \ell_3 \parallel \ell_2 : \star \parallel \ell_3 : \alpha_c \alpha_r,\ \varphi'[\alpha_w]
\end{aligned}$$

The choice of the service ℓ_2 for the request r_1 results in downloading an applet from site ℓ_2. This seems correct, until the plan chooses the service ℓ_3 to execute the applet. The computation aborts because the applet provided by ℓ_2 attempts to violate the policy φ', that becomes active after the service ℓ_2 has returned.

As a matter of fact, the tree-shaped structure of planned selections makes it difficult to determine the plans under which a history expression is valid. To cope with this problem, we have devised a static analysis that "linearizes" such a tree structure into a set of history expressions, forming an equivalent planned selection $\{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$, where no H_i has further planned selections.

In our running example, we find that H is equivalent to the following H':

$$\begin{aligned}
H' = \{ & r_1[\ell_1] \mid r_2[\ell_3] \triangleright \ell_3 : \alpha_c \cdot \varphi'[\varphi[\alpha_r]], \\
& r_1[\ell_2] \mid r_2[\ell_4] \triangleright \ell_4 : \alpha_r \cdot \alpha_w, \\
& r_1[\ell_1] \mid r_2[\ell_4] \triangleright \ell_4 : \varphi[\alpha_r], \\
& r_1[\ell_2] \mid r_2[\ell_3] \triangleright \ell_3 : \alpha_c \cdot \varphi'[\alpha_r \cdot \alpha_w]\}
\end{aligned}$$

Every element of H' clearly separates the plan from the associated abstract behaviour. This piece of behaviour has no further plans within, and so it is easier to model-check its validity.

For instance, under the plan $r_1[\ell_1] \mid r_2[\ell_3]$, the overall abstract behaviour is $\alpha_c \cdot \varphi'[\varphi[\alpha_r]]$ at site ℓ_3. As already seen, the first two plans in H' are viable, while the others give rise to non-valid behaviour. The plan $r_1[\ell_2] \mid r_2[\ell_4]$ is *not* viable, because the policy φ would be violated when the obtained code f is run on a non certified site; instead, the plan $r_1[\ell_2] \mid r_2[\ell_3]$ would violate φ'.

Given a history expression H, we obtain its linearization as follows. First, we define an equational theory of history expressions: an equation $H \equiv H'$ means that H and H' represent the same histories, under all plans. Roughly, our equations say that each history expression $\mathcal{C}(H)$ is equivalent to some planned selection H'. For instance, when $\mathcal{C}(H) = \varphi[H]$, we have that:

$$\varphi[\{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}\ \equiv\ \{\pi_1 \triangleright \varphi[H_1] \cdots \pi_k \triangleright \varphi[H_k]\}$$

If $\mathcal{C}(H)$ is already a planned selection, then either it is linear, or it has one level of nesting more than H'. For instance, if $\mathcal{C}(H) = \{\pi_0 \triangleright H\}$, then:

$$\{\pi_0 \triangleright \{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}\}\ \equiv\ \{\pi_0 \mid \pi_1 \triangleright H_1 \cdots \pi_0 \mid \pi_k \triangleright H_k\}$$

When oriented from left to right, these equations give rise to a rewriting system that is easily proved finitely terminating and confluent – up to the equational laws (commutativity, associativity, idempotence, and zero) of the algebra of plans. The resulting planned selection is linear.

For instance, the linearization of the history expression H inferred for our client is constructed as follows:

$$\begin{aligned}
&\{r_2[\ell_3] \rhd \ell_3 : \alpha_c \cdot \varphi'[\{r_1[\ell_1] \rhd \varphi[\alpha_r], r_1[\ell_2] \rhd \alpha_r \cdot \alpha_w\}] \\
&\quad r_2[\ell_4] \rhd \ell_4 : \{r_1[\ell_1] \rhd \varphi[\alpha_r], r_1[\ell_2] \rhd \alpha_r \cdot \alpha_w\}\} \\
\equiv\ &\{r_2[\ell_3] \rhd \ell_3 : \alpha_c \cdot \{r_1[\ell_1] \rhd \varphi'[\varphi[\alpha_r]], r_1[\ell_2] \rhd \varphi'[\alpha_r \cdot \alpha_w\}]] \\
&\quad r_2[\ell_4] \rhd \{r_1[\ell_1] \rhd \ell_4 : \varphi[\alpha_r], r_1[\ell_2] \rhd \ell_4 : \alpha_r \cdot \alpha_w\}\} \\
\equiv\ &\{r_2[\ell_3] \rhd \{r_1[\ell_1] \rhd \ell_3 : \alpha_c \cdot \varphi'[\varphi[\alpha_r]], r_1[\ell_2] \rhd \ell_3 : \alpha_c \cdot \varphi'[\alpha_r \cdot \alpha_w\}]] \\
&\quad r_2[\ell_4] \rhd \{r_1[\ell_1] \rhd \ell_4 : \varphi[\alpha_r], r_1[\ell_2] \rhd \ell_4 : \alpha_r \cdot \alpha_w\}\} \\
\equiv\ &\{r_1[\ell_1] \mid r_2[\ell_3] \rhd \ell_3 : \alpha_c \cdot \varphi'[\varphi[\alpha_r]],\ r_1[\ell_2] \mid r_2[\ell_4] \rhd \ell_4 : \alpha_r \cdot \alpha_w, \\
&\quad r_1[\ell_1] \mid r_2[\ell_4] \rhd \ell_4 : \varphi[\alpha_r],\ r_1[\ell_2] \mid r_2[\ell_3] \rhd \ell_3 : \alpha_c \cdot \varphi'[\alpha_r \cdot \alpha_w]\}
\end{aligned}$$

The technical role of linearization is unveiled by the following theorem, that will enable us to detect the viable plans for service composition.

Theorem 2. *If $H = \{\pi_1 \rhd H_1 \cdots \pi_k \rhd H_k\}$ is linear, and H_i is 0-valid for some $i \in 1..k$, then H is π_i-valid.*

Summing up, we extract from an client e a history expression H, we linearize it into $\{\pi_1 \rhd H_1 \cdots \pi_k \rhd H_k\}$, and if some H_i is valid, then we can deduce that H is π_i-valid. By Theorem 1, the plan π_i safely drives the execution of e, without resorting to any run-time monitor.

Extracting viable plans II: Verifying validity

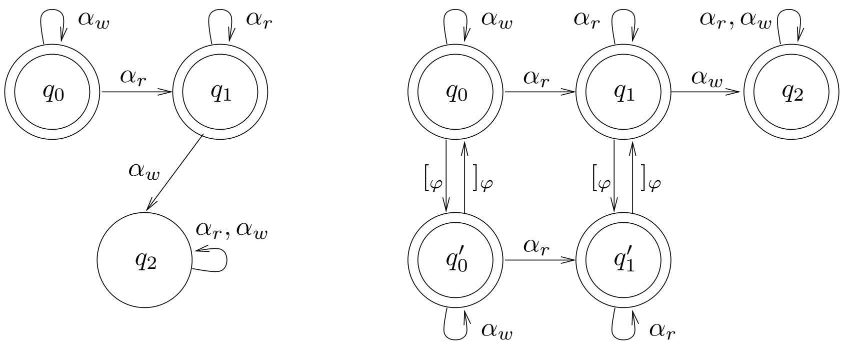
To verify the validity of history expressions that have no planned selections, it suffices to apply the verification technique of [7], briefly described below. Our technique consists in smoothly transforming history expressions in procesess of Basic Process Algebras (BPAs), and in model checking them with Finite State Automata (FSA). The standard decision procedure for verifying that a BPA process p satisfies a (ω-regular) property φ amounts to constructing the pushdown automaton for p and the Büchi automaton for the negation of φ. Then, the property holds if the (context-free) language accepted by the conjunction of the above, which is still a pushdown automaton, is empty. This problem is decidable, and several algorithms and tools show this approach feasible [20]. Since our execution histories are always finite and our properties are regular, it turns out that we can simplify this procedure by using FSA, instead of Büchi automata.

However, our notion of validity is *non-regular*, because of the arbitrary nesting of framings. As an example, language denoted by $H = \mu h.\, \alpha + h \cdot h + \varphi[h]$ is context-free and non-regular, because it contains unbounded nesting of framings (technically, it is equivalent to the language of balanced parentheses). Since context-free languages are not closed under intersection, the emptiness problem is undecidable. To apply the procedure sketched above, we then manipulate history expressions in order to make validity a regular property.

The intuition is that non-regularity is a consequence of *redundant framings*, i.e. vacuous nesting of the same framing. For example, the history $\eta = \alpha\varphi[\alpha'\varphi'[\varphi[\alpha'']]]$ has an inner redundant safety framing φ around α''. Since α'' is already under the scope of the outermost φ-framing, it happens that η is valid if and only if $\alpha\varphi[\alpha'\varphi'[\alpha'']]$ is valid.

In [8], we have defined a validity-preserving transformation that, given a history expression H, yields a $H\downarrow$ that does not generate redundant safety framings. Also, for each policy φ, we have defined a formula $\varphi_{[]}$ to be used in verifying the validity (w.r.t. φ) of histories with no redundant framings.

For instance, consider again φ' saying that no event α_w can occur after α_r. The finite state automata enforcing φ and $\varphi_{[]}$ are shown below, where the special events $[_\varphi$ and $]_\varphi$ denote the opening and closing of the scope of φ, respectively. It is immediate to check that the history $[_\varphi\alpha_r]_\varphi\alpha_w$ is accepted by $A_{\varphi_{[]}}$, while $\alpha_c[_\varphi\alpha_r\alpha_w]_\varphi$ is not.



Validity of history expression H with no planned selections can be decided by showing that the BPA generated by the regularization of H (written $BPA(H\downarrow)$) satisfies a suitably constructed regular formula.

Together with Theorem 1, this dispenses us from using an execution monitor to enforce the security policies on demand.

Theorem 3. *A history expression H with no planned selections is* 0*-valid iff:*

$$[\![BPA(H\downarrow)]\!] \models \bigwedge_{\varphi\in H} \varphi_{[]}$$

Back to our running example, we have that:

$$\begin{aligned}
\alpha_c[_{\varphi'}[_\varphi\alpha_r]_\varphi]_{\varphi'} \models \varphi_{[]} \wedge \varphi'_{[]} &\implies r_1[\ell_1] \mid r_2[\ell_3] \ \ \text{viable,}\\
\alpha_r\alpha_w \models tt &\implies r_1[\ell_2] \mid r_2[\ell_4] \ \ \text{viable,}\\
[_\varphi\alpha_r]_\varphi \not\models \varphi_{[]} &\implies r_1[\ell_1] \mid r_2[\ell_4] \ \ \textit{not}\ \text{viable}\\
\alpha_c[_{\varphi'}\alpha_r\alpha_w]_{\varphi'} \not\models \varphi'_{[]} &\implies r_1[\ell_2] \mid r_2[\ell_3] \ \ \textit{not}\ \text{viable}
\end{aligned}$$

Beyond simple plans

Recall that so far we have only considered *simple* plans that associate a single service with each request. Indeed, planning service composition can be more complex, as we will show in a while.

Assume first that the two requests r_1 and r_2 in the client are repeatedly performed in a loop. Then, suppose that a new service for r_2 is discovered at site ℓ_5, offering to run the code f without any constraints. Note that, if we stick to simple plans, then we must choose once and for all one among the viable plans, i.e. $r_1[\ell_1] \mid r_2[\ell_3]$, $r_1[\ell_2] \mid r_2[\ell_4]$, and $r_1[\ell_2] \mid r_2[\ell_5]$. Consequently, at each iteration of the loop the *same* service is taken for the request r_2. To be more flexible (i.e. in case the service chosen for r_2 becomes unavailable), we would like to accept as valid also the plan $r_1[\ell_2] \mid r_2[\ell_4, \ell_5]$, where r_2 can be served by either ℓ_4 or ℓ_5. This is just an example of *multi-choice* plans, where a request can be resolved by a set of services. In our running example, this has the advantage of permitting to select for r_2 between ℓ_4 and ℓ_5 at each iteration of the loop.

Consider now another slight extension of our example, where the client is billed for the services it has invoked. To do that, assume that an argument g is passed to the request r_1, to invoke a billing service through a request r_3, and so let the code provider invoice the customer ℓ_0 for the service. The same function g is also passed later on the service which will actually run the code f, to charge ℓ_0 for the cost of the execution.

A billing service acts as a function that takes as input an invoice (of some type τ', immaterial here) and delivers back a payment certification, i.e. a function of type $\tau' \xrightarrow{\alpha_{paid}} \tau'$ that generates α_{paid} to signal successful transaction. Let $\tau_b = \tau' \to (\tau' \xrightarrow{\alpha_{paid}} \tau')$ be the type of billing services. Then, the request types of r_1 and r_2 would have the following form:

$$\rho_1 = \tau \times \tau_b \xrightarrow{\psi} (\tau \to \tau) \qquad \rho_2 = (\tau \to \tau) \times \tau_b \xrightarrow{\psi} \tau$$

where the property ψ on demand requires that payment is accomplished before the control returns back to the client.

Assume now that two billing services ℓ_6 and ℓ_7 are discovered in the network. The service ℓ_6 can be used by certified users only, while ℓ_7 imposes no constraints. Clearly, the service which provides the code and the one which runs it can choose different billing services. However, neither simple nor multi-choice plans can render adequately this situation. The simple plan (yet ill-formed) that seems to solve the problem is $r_1[\ell_1] \mid r_2[\ell_3] \mid r_3[\ell_6] \mid r_3[\ell_7]$. However, this plan cannot express the linkage between the choice for r_3 within code providers, and that within code executers.

We therefore extend plans to keep track of dependencies among choices. The *dependent* plan $r_1[\ell_1.r_3[\ell_7]] \mid r_2[\ell_3.r_3[\ell_6]]$ is viable: the request r_3 is resolved with ℓ_7 within the service ℓ_1 chosen for r_1, while it is resolved with ℓ_6 within the service ℓ_3 chosen for r_2. In [9] we have shown how to extract viable plans from history expressions, also in the case of multi-choice and dependent plans.

3 Conclusions

A static approach has been proposed to study secure orchestration of services. We have surveyed a distributed calculus with an explicit notion of location and of located executions. Our calculus has primitives for enforcing local security policies, and for invoking services that respect given security requirements.

We have devised a way of statically constructing the plans that drive successful, secure executions. The actual histories that can occur at run-time are over-approximated by a type and effect system. These approximations are then model-checked to find the plans that guarantee secure executions, without the need of execution monitoring.

Acknowledgments

Research partially supported by the EU, within the FETPI Global Computing, Project IST-2005-16004 Sensoria (Software Engineering for Service-Oriented Overlay Computers).

References

1. M. Abadi and C. Fournet. Access control based on execution history. In *Proc. 10th Annual Network and Distributed System Security Symposium*, 2003.
2. R. Akkiraju et al. *Web Service Semantics.* WSDL-S technical note (version 1.0), 2005.
3. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications.* Springer-Verlag, 2004.
4. S. Anderson et al. *Web Services Trust Language (WS-Trust)*, 2005.
5. T. Andrews et al. *Business Process Execution Language for Web Services (BPEL4WS), Version 1.1*, 2003.
6. B. Atkinson et al. *Web Services Security (WS-Security)*, 2002.
7. M. Bartoletti, P. Degano, and G. L. Ferrari. Enforcing secure service composition. In *Proc. 18th Computer Security Foundations Workshop (CSFW)*, 2005.
8. M. Bartoletti, P. Degano, and G. L. Ferrari. History based access control with local policies. In *Proc. Fossacs*, 2005.
9. M. Bartoletti, P. Degano, and G. L. Ferrari. Plans for service composition. In *Workshop on Issues in the Theory of Security (WITS)*, 2006.
10. M. Bartoletti, P. Degano, and G. L. Ferrari. Types and effects for secure service orchestration. In *To appear in Proc. 19th Computer Security Foundations Workshop (CSFW)*, 2006.
11. K. Bhargavan, R. Corin, C. Fournet, and A. D. Gordon. Secure sessions for web services. In *Proc. ACM Workshop on Secure Web Services*, 2004.
12. K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. In *Proc. ACM Symposium on Principles of Programming Languages*, 2004.
13. E. Bonelli, A. Compagnoni, and E. Gunter. Typechecking safe process synchronization. In *Proc. Foundations of Global Ubiquitous Computing*, 2004.
14. D. Box et al. *Simple Object Access Protocol (SOAP) 1.1.* WRC Note, 2000.

15. D. Box et al. *Web Services Policy Framework (WS-Policy)*, 2002.
16. A. Brogi, C. Canal, and E. Pimentel. Behavioural types and component adaptation. In *Proc. AMAST*, 2004.
17. A. Brogi and R. Popescu. Towards semi-automated workflow-based aggregation of web services. In *Proc. ICSOC*, 2005.
18. R. Chinnici, M. Gudgina, J. Moreau, and S. Weerawarana. *Web Service Description Language (WSDL), Version 1.2*, 2002.
19. F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarane. The next step in web services. *Communications of the ACM*, 46(10), 2003.
20. J. Esparza. On the decidability of model checking for several μ-calculi and Petri nets. In *Proc. 19th Int. Colloquium on Trees in Algebra and Programming*, 1994.
21. D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, 1986.
22. A. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. In *Proc. IEEE Computer Security Foundations Workshop*, 2002.
23. D. Gorla, M. Hennessy, and V. Sassone. Security policies as membranes in systems for global computing. In *Proc. FGUC*, 2004.
24. K. Honda, V. Vansconcelos, and M. Kubo. Language primitives and type discipline for structures communication-based programming. In *Proc. ESOP*, 1998.
25. R. Khalaf, N. Mukhi, and S. Weerawarana. Service oriented composition in BPEL4WS. In *Proc. WWW*, 2003.
26. A. Lazovik, M. Aiello, and R. Gennari. Encoding requests to web service compositions as constraints. In *Constraint Programming CP*, 2005.
27. S. B. Mokhtar, N. Georgantas, and V. Issarny. Ad hoc composition of user tasks in pervasive computing environment. In *Software Composition*, 2005.
28. F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design*, 1999.
29. M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matchmaking of web services capabilities. In *First International Semantic Web Conference on The Semantic Web*, 2002.
30. M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE*, 2003.
31. M. Papazouglou and D. Georgakopoulos. Special issue on service oriented computing. *Communications of the ACM*, 46(10), 2003.
32. P. Rajasekaran, J. A. Miller, K. Verma, and A. P. Sheth. Enhancing web services description and discovery to facilitate composition. In *Semantic Web Services and Web Process Composition*, 2005.
33. P. Sewell and J. Vitek. Secure composition of untrusted code: box-π, wrappers and causality types. *Journal of Computer Security*, 11(2), 2003.
34. M. Stal. Web services: Beyond component-based computing. *Communications of the ACM*, 55(10), 2002.
35. J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 2(111), 1994.
36. P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *Proc. ISWC*, 2004.
37. A. Vallecillo, V. Vansconcelos, and A. Ravara. Typing the behaviours of objects and components using session types. In *Proc. of FOCLASA*, 2002.
38. W. Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6), 2003.
39. W3C. *UDDI Technical White Paper*, 2000.

40. T. Woo and S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Security and Privacy*, 1993.