Using the new SmartFrog Security -DRAFT-

Antonio Lain
HP Labs, Bristol

13th January 2006

Abstract

A new version of SmartFrog provides a much more flexible security model. In this document we will give an overview of this model and how it is enforced. Then, we will provide a few simple examples that are trying to motivate the need for this complexity; hopefully, encouraging the reader to start using the new features. Finally, we suggest more advanced scenarios that give hints of its full potential.

1 About this Document

This document tries to be a self-contained, "gentle" introduction to the new SmartFrog security. It assumes a reasonable working knowledge of "standard" SmartFrog [3], i.e., language, component model, and deployment engine. It does not require experience with the previous SmartFrog security, but it is critical certain familiarity with basic security concepts, e.g., authentication, authorization, certificates, PKI,...

The rest of this document is structured as follows: Section 2 gives an overview of what we want to achieve with the new security, i.e., its rationale, goals, assumptions, expected threats... Section 3 shows how to describe security relationships between components in the new model. After discussing how to describe relationships, we cover in Section 4 the mechanisms that we use to enforce them. Then, we start describing examples that will be included in the new code release; we separate them in two sections: Section 5 contains "basic" examples that require little understanding of the model, and Section 6 includes more "advanced" cases that start to show the power of the framework. Finally, we give our conclusions in Section 7.

This is a long document, do I need to read everything?... If you just want to get “the basics”, you could skip most of Section 4 and Section 6. However, it is only in the “advanced” sections where you will discover the potential of the new features...

2 Introduction

2.1 What’s wrong with the previous security?

The previous security model was based on a single trusted community of “equal” peers; that is, you can either be “inside” or “outside” my community. If you are “inside” you can do “almost anything”, e.g., deploy new components, change loadable resources such as classes or descriptions, interact with remotely deployed components, and so on. If you are “outside” you can do “almost nothing”, since most remote interactions will use authenticated secure channels, and loadable resources are always signed by someone in the community.

The main benefit of this approach is simplicity; no change to descriptions or java code is required to activate the security [1]. Also, the “set-up” of security is reasonably straightforward, with the issuing of new credentials and the signing of jar files being automated by Ant [2] build files.

Unfortunately, the current model is sometimes too simplistic. For example, we would like to interact with less trusted peers without giving them full privileges over our “community”. Moreover, if one trusted member is compromised the whole community is in danger, and this limits drastically the security of a large community. Also, the top-down process to decide who is trusted does not match well federated environments that do not have a single root of trust and rely on (constrained) delegation for scalability.

Most existing management systems for distributed components tend to centralise security policy decisions in order to limit exposure to a compromise of a single peer. However, this is unworkable in a fully distributed framework like SmartFrog where everything happens through peer to peer interactions that are not mediated.

So this was our main challenge for the new security features. Describe and enforce more fine-grained security relationships, but without affecting the “peer-to-peer” nature of SmartFrog interactions. Let’s discuss in more detail some of the properties that we wanted to achieve.

[1] However, you are only “safe” if the deployed components comply with certain requirements (see Section 2.3).

2.2 Goals for the new security features

Explicit customise the security relationships of the components that we are deploying by modifying SmartFrog descriptions.

Dynamic security domains partition dynamically, i.e., through the deployment of components, a set of peers so that each group of peers can enforce its own security policy. For example, they only trust members of its own group or members of group X.

Cooperative deployment ensure that peers can cooperate to achieve a deployment without having to fully trust each other.

Lazy customisation modify "on-the-fly" a deployment description by less trusted peers but in a safe manner. For example, if one peer is responsible for dynamic allocation of tasks to other peers, he can change location attributes in a description to redirect this task to a different node, but he should not be able to modify the attribute that defines the java class implementing that task.

Controlled delegation enable autonomous changes in a security domain, e.g, change membership in a trusted group of peers or create a less trusted subdomain. This implies that a peer should be able to (partially) delegate his authority to other peers.

Federated allow peers in the same security domain with credentials issued by different authorities that do not fully trust each other.

Non-hierarchical establish relationships among security groups that do not have a single root of authority common to all of them. Moreover, authorization should not rely on a globally unique naming scheme that identify peers.

Back-wards compatible minimise changes to existing descriptions and components when the new features are not needed. Also, be able to mimic the behaviour of the previous security model.

Unfortunately, there are trade-offs among these properties. The flexibility of lazy customisation of descriptions can increase the level of trust required between cooperating peers. The frequency of membership changes in a group can affect how effectively we can control delegation. The lack of a single authority that defines a group can make more difficult to enforce a consistent security policy. The requirement of minimal impact on existing components limits the "explicitness" of our descriptions...

Our approach is to follow the SmartFrog "framework dogma". Allow the end user to make these trade-offs by writing descriptions and components that fit his/her needs. For example, if your application does not need federation or non-hierarchical relationships, you could "hardwire" a "root public key" in your descriptions. If no lazy customisation is needed, never use the keyword "SIGNED" in a description. If there is no need of dynamic groups, do not create components that issue credentials. If only a single trust community is needed, just change the description of the default security domain, i.e., in `ssd.sf`, to do so.

The drawback of the "framework dogma" is a potentially steep learning curve. Trying to use all these features in early stages is a recipe for security disaster. We recommend to start by re-using the templates and examples provided, and to leverage advanced features as they are needed. We are also working on some high level tools that will ease the security set-up and the annotation of descriptions.

2.3 Security assumptions

Our security assumptions have not changed that much since the previous model:

- Local interactions among components, i.e., deployed in the same JVM (Java Virtual Machine), are always considered "safe"; we just worry about remote interactions. The rationale is that, in most cases, SmartFrog components perform operations that need "high privileges", e.g., installing application software in the local disk, and it is not realistic to impose an "applet-browser" style of separation.

- Installation and protection of the SmartFrog core classes and security credentials is ensured by a "safe" boot-strapping process. For example, we could have an "off-line" mechanism to distribute keys and core SmartFrog classes, and rely on the local OS (Operating System) to protect them afterwards.

- The signed software implementing the deployed components is "trusted", i.e., it "cooperates" with the SmartFrog security mechanisms. For example, components rely on RMI for remote interaction and they do not use their own "unsafe" class loaders.

- The security of external applications deployed by SmartFrog, i.e., software that is not a SmartFrog component, is beyond the scope of SmartFrog security. SmartFrog could help to boot-strap safely the security mechanisms of these applications, but we do not interfere with them after that...

- Our network is not trusted and we always use encryption and PKI-based authentication to create end-to-end secure channels between peers.

- If a principal can deploy **any**[2] description on a node or modify implementation classes of components before they are deployed, he "owns" the node. For example, he could deploy software that reads the local private key or executes arbitrary code with the same OS privileges as the JVM process. Therefore, in most practical cases the "owner" has "root" or "administrator" OS privileges in that node, since application management typically relies on that level of privileges.

- The integrity of dynamically loaded resources is guaranteed by signing the jar files that contain them. However, by default, their confidentiality is not guaranteed since web servers hosting them are not always trusted.

- Deploying components could dynamically change the security policy that a node tries to enforce. We do not guarantee that the new policy will be enforced for previously deployed local components, since we aggressively cache policy decisions. Moreover, we cannot guarantee that side-effects of the new policy will not affect the behaviour of "old" deployed components either. Therefore, drastic changes to policy should first terminate currently deployed components to ensure consistency. However, if the new policy is less restrictive, and it is also acceptable for already deployed components, we could sometimes avoid the "reset"...

2.4 Threats

The threats that we want to protect SmartFrog against are common to many distributed management systems:

- Passive or active network attacks that break the confidentiality or integrity of the communications among our components.

- Deployment of malicious software either by changing component descriptions or by modifying classes that implement components.

- Unauthorised changes to the configuration of existing components.

- Propagation of a single node compromise to a whole "community" of nodes. Note that even if the initial compromise is not caused by SmartFrog software, we want to limit the use by the attacker of the SmartFrog infrastructure to spread damage to other nodes.

[2]Note that a description signed by an "owner" could be deployed by a less trusted principal (see Section 4.1).

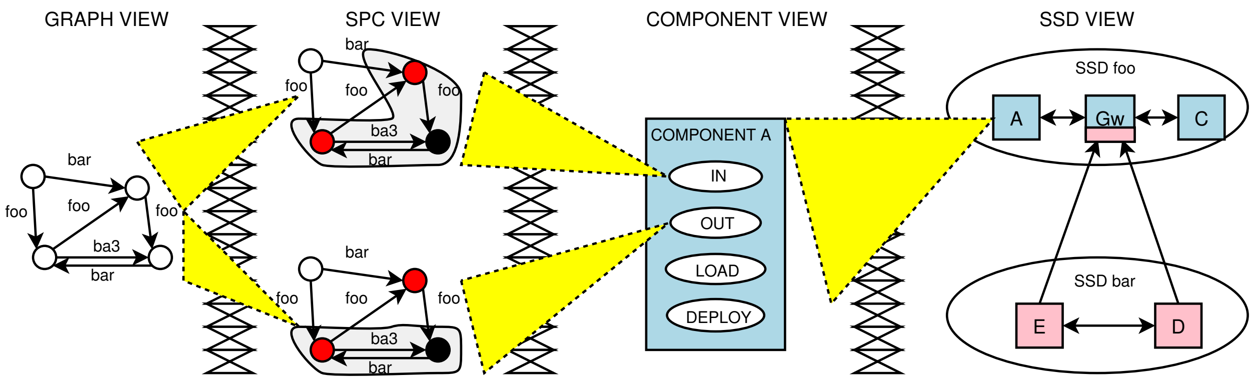


Figure 1: Overview of the security model

- Denial-of-service of the management infrastructure. Unfortunately, unless the management network is isolated, there is little SmartFrog can do against a “low-level”, “brute force” network attack...

3 Describing Security Relationships

We propose a new security model that represents fine-grained, federated security relationships between distributed components, that can be enforced by using SmartFrog deployment mechanisms. Figure 1 shows us the different views that we use in our model to describe these relationships:

Graph View Nodes in the graph represent principals, i.e., public keys. Labelled edges represent local bindings between principals. We have a way to propagate parts of this graph safely by using certificates (see Section 3.1).

Simple Path Constraint (SPC) View Restricts the underlying graph by using regular constraints on labels connected through simple paths (see Section 3.2).

Component View Associates multiple SPCs to a component so that we can control its external interactions (see Section 3.3).

SmartFrog Security Domain (SSD) View Groups components with similar policies in domains, and represents controlled interactions among them through “gateways” (see Section 3.4).

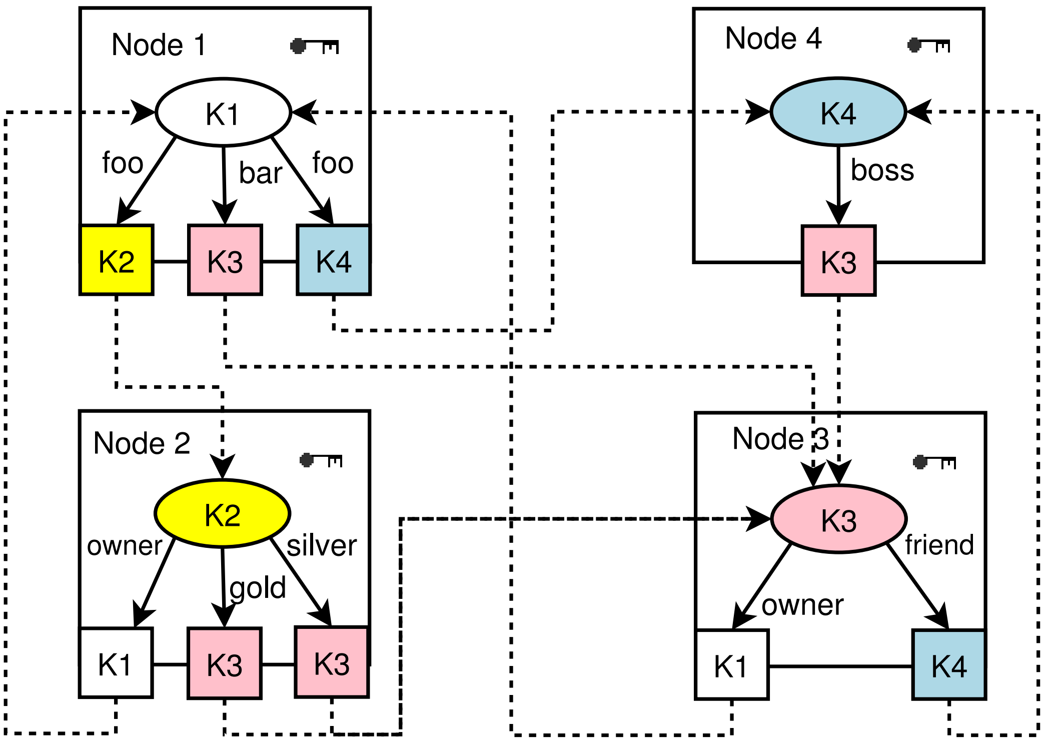


Figure 2: An example of representing relationships in the new security model

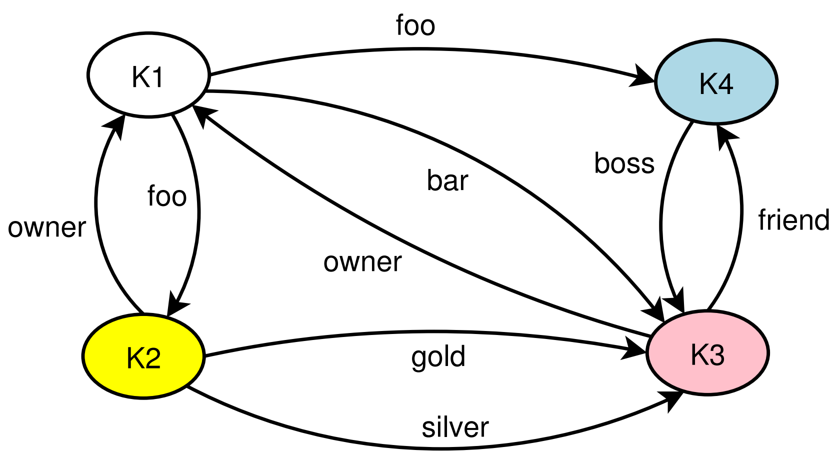


Figure 3: Graph representation of example in Figure 2

3.1 The basic graph model

Figure 2 shows an example of the underlying model that we use to represent security relationships using the new features. We assume that each principal is identified by a public/private key pair, where in this context a principal could be a running SmartFrog daemon, a user that is trying to deploy a description, or an off-line entity that we use to sign jar files. Each principal maintains its own set of local bindings that associate a label to the public key of other principal. These labels do not need to be locally unique and the same public key can have more than one local binding; this implies that labels are more like local properties associated with other principals rather than a mechanism to identify them, i.e., if we need to authenticate them uniquely we should use their public keys.

In this context we can view a certificate as a safe way that a principal has to make visible one of its local bindings to the "world"; in contrast to the conventional interpretation of certificates, i.e., asserting the "global" identity of other principal. In fact, even though we embed this binding in a valid X.509 certificate, the critical information derived from it is the public keys of the issuer and target principals and the label associated with this binding, and not the irrelevant DN (Distinguished Name) of the target[3].

Figure 3 shows a convenient way to represent all these bindings by using a directed graph with labelled edges. The mapping is trivial, with nodes of the graph associated with principals and edges representing local bindings with corresponding labels. In the rest of this document the words "graph edge", "local binding" or "certificate" mean roughly the same thing. Similarly, we interchange informally

[3]We use a string representation of the hashed public key to generate DNs.

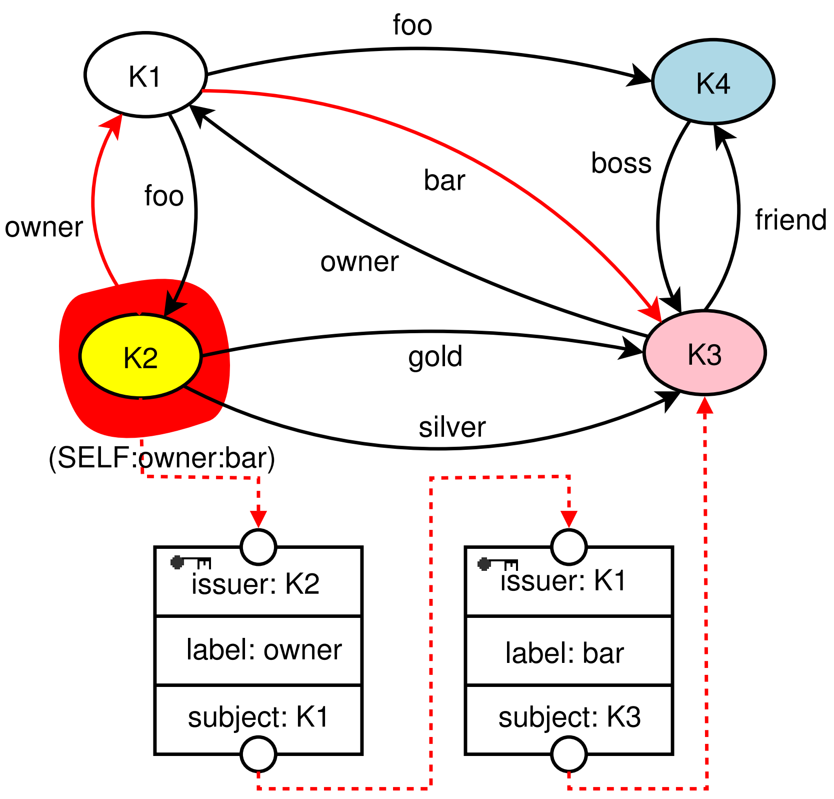


Figure 4: K3 proves (SELF:owner:bar) to K2

"graph node", "principal" or "peer".

One of the most powerful features of the model (inspired by SDSI [6] linked local name spaces) is that a principal can use the local bindings of other principals for its own authorization decisions. This is done by traversing a path in the graph that uses these principals as intermediate nodes, and concatenating the corresponding labels, i.e., forming a "path label". For example, in Figure 4, `K2` could use `K1`'s local bindings with label `bar` by concatenating labels `owner` and `bar` to form `"(SELF:owner:bar)"`[4]. When `K3` wants to prove to `K2` that it validates `"(SELF:owner:bar)"` it will need to show knowledge of its corresponding private key and attach the certificate issued by `K1` that links its key to `bar`. Then, `K2` can (implicitly) complete a certificate chain from itself to `K1` (for `owner`) and from `K1` to `K3` (for `bar`), that corresponds to a simple path in the corresponding graph from `K2` to `K3` (via `K1`). Note that we always force paths to be "simple", i.e., no cycles, and this increases compatibility with X.509 certificate chain validation

[4]We use the keyword `SELF` to identify the public key of whoever is evaluating the path label. If we want to make a path label absolute we just replace `SELF` by a public key, e.g., `K2`.

and also makes rules easier to understand.

Clearly, if we use the local bindings of other principal for authorization we are implicitly delegating some authority to him. He could create a new binding with the same label and associate it with any other principal. He could also create multiple public/private key pairs for fictitious principals and extend the path label arbitrarily. For this reason, if a principal can prove a non empty "prefix" of a path label, we assume that he can prove that path label. For example, since `SELF` can always be proved by the public/private key of the principal checking that path label, he can also prove any path label that starts by `SELF`; therefore, "local interactions" with these path labels are always permitted.

3.2 Simple Path Constraint (SPC)

We could directly use a path label for authorization purposes by adding it to an access control list (ACL). In this case, only a principal that proves the path label, using the mechanism described above, will be able to perform the requested action. However, it can be very tedious to enumerate all possible satisfactory path labels. Instead, we describe sets of valid path labels using a subset of regular path expressions[5], i.e., a Simple Path Constraint (SPC). In particular, the matching elements of an SPC, separated by ":", can be:

`<SELF>` matches the public key of the principal that is evaluating the ACL (keyword "`SELF`").

`<PUBKEY>` matches a principal with a given public key. We represent the public key with a character "=" followed by a string base-64 representation of a self-signed X.509 certificate.

`<PATTERN>` applies a string pattern matching expression to a single edge label. Currently, we only support simple patterns that contain "`*`".

`<NOBODY>` matches no binding (keyword "`NOBODY`").

`<DOTS>` matches a sequence of arbitrary labels, i.e., graph transitive closure ignoring labels. An empty sequence is also a valid match (keyword "`...`").

A valid SPC should either just contain the element `<NOBODY>` or `<DOTS>`, or start by `<SELF>` or `<PUBKEY>` followed by a (possibly empty) sequence of `<PATTERN>` elements and (possibly) finished by a `<DOTS>` element. This is described in BNF form as follows:

[5]We need to restrict regular expressions because, when combined with our simple path requirement, it leads to decision algorithms of high complexity [5].

```
 <NOBODY>  | <DOTS> |
 ((<SELF> | <PUBKEY> ) : (<PATTERN>)* : <DOTS>? ))
```

For example, possible valid SPCs are:

```
(SELF:...)
(=werweWWsdwerrewr2asdsdd:foo*:bar_*3)
(NOBODY)
(SELF:foo:...)
(SELF)
(=sddsasdsdsadasdd3323423424)
```

but these are not valid:

```
(SELF:=assds323432424234)
(SELF:...:foo*1)
(SELF:NOBODY)
(foo*:bar21)
```

So how do we know if a path label obtained from an authenticated simple traversal satisfies an SPC?. The detailed algorithm is described somewhere else [4], but a simpler explanation is that we try to match in sequence each label in the target path label with its corresponding matching element in the SPC. If a label does not match we stop with "false", otherwise we continue with the next one until either we have no more elements in the SPC (return "false"), or no more elements in the path label (return "true" if we have already matched a prefix, otherwise "false"). Note that the matching elements `<NOBODY>` and `<DOTS>` return immediately with values "false" and "true".

Let's look at some examples based on the set-up in Figure 3:

```
K1 evaluates (SELF:foo) -> K1, K2 and K4 are OK
K1 evaluates (K4:*)  -> K4 and K3 are OK (no K1!)
K1 evaluates (SELF:...) -> K1, K2, K3 and K4 are OK
K1 evaluates (SELF:b*ar:...) -> K1, K3 and K4 are OK
                              (no K2 due to cycles!)
K2 evaluates (SELF:foo) -> K2 is OK
K2 evaluates (SELF:gold*) -> K2 and K3 are OK
K2 evaluates (NOBODY) -> none are OK
K4 evaluates (SELF:...) -> K4, K3, K1 and K2 are OK
```

3.3 Component Constraints

After describing in the previous section how to represent access control using an SPC, we will explain how we are going to use SPCs to secure components. A deployed SmartFrog component is an object that has interfaces with methods that can be called remotely using RMI[6]. It also can obtain stubs for other remote objects and invoke methods on them. Moreover, the java classes or other resources needed to create this component might be dynamically downloaded from untrusted sources, e.g., a web server. Finally, components that implement the `Compound` interface have remotable methods that can be used to deploy descriptions parented by them. Clearly, all these operations could involve security risks and an access control policy needs to be enforced.

Each component declared in a SmartFrog description has a "place holder" attribute where we collect all the security sensitive attributes associated with that component, i.e., `sfSSD`. The default value for this attribute is obtained from its parent component if it has not been explicitly defined (see file `org/smartfrog/sfcore/prim.sf`). At the top level of every description we set a default value for `sfSSD` with the value of the attribute *`SSD`*. The default value for `SSD` is specified in the file :

`org/smartfrog/sfcore/security/sf/ssd.sf`

and the intention is that, by changing that file or replacing the definition of `SSD`, we can affect the default security policy of all the components that do not need a special policy. Let's look at the default contents of `ssd.sf`:

```
// Default domain is "just trust myself"
SSD_RESTRICTED extends LAZY {
  ScopeACL "(SELF)";
  CallInACL "(SELF)";
  CallOutACL "(SELF)";
  CodeBaseACL "(SELF:owner)";
  DeployDescriptionACL "(SELF:owner)";
}
//Default is the restricted model.
SSD ATTRIB SSD_RESTRICTED;
```

So in a good paranoid security tradition the default policy is pretty tight. Let's look at the inner `sfSSD` attributes in more detail:

`ScopeACL` defines the requirements on the security domain in which this component should be deployed (we discuss this attribute in the next section).

[6]We can state in the description that a component should not be "exported", i.e., made accessible remotely through RMI. However, this case is not a problem for our security model, since we only worry about remote interactions.

`CallInACL` describes who can call the remote methods of this component. In this case only local incoming calls are allowed.

`CallOutACL` describes the requirements on the other remote objects that we invoke their methods. In this case only local outgoing calls are allowed.

`CodeBaseACL` describes who can sign the jar files from which we load classes and resources to deploy this component. In this case we only allow `owner` to sign jar files, which is the default name used in Ant build files, as we will describe in Section 5.

`DeployDescriptionACL` describes who can sign descriptions that this component will deploy and parent - note that only a component that implements the `Compound` interface has that capability anyway -. Here we only allow `owner`, the default name that the Ant build files use, to sign descriptions[7].

In an ideal world you could specify a different security policy for each component, deploy them in the same SmartFrog daemon, and ensure that policies are independently enforced and do not affect each other. Unfortunately, this is not the case in SmartFrog, because there is no strong separation between locally deployed components (see Section 2.3), and we have to be very careful on deploying together components that enforce consistent policies.

We delay to the implementation sections details on how we are dealing with policy conflicts: Section 4.2 discusses class loading issues; Section 4.3 describes conflicts in remote interactions, i.e., RMI. However, we can anticipate that, depending on the nature of these conflicts, we could face situations in which it is difficult to predict the policy currently enforced, or we could affect the behaviour of previously deployed components. Therefore, it is recommended to associate a single security role to a daemon and do a full reset when we need to change significantly that role.

3.4 SmartFrog Security Domain (SSD)

If we need to track the security policy of each component individually, complexity will grow fast, and it will be nearly impossible to understand the "big picture" or to detect policy conflicts. Instead, we would like to group components that require a similar security policy, and deploy them on collections of SmartFrog daemons that have the right credentials to support this common policy. Then, we can reason about a deployed system based on the relationships between these groups, rather

[7]Note that this signing is done after all the SmartFrog parser phases, and it has no relation to the signatures associated with jar files that `CodeBaseACL` specifies. We will try to clarify this further in Section 4.1.

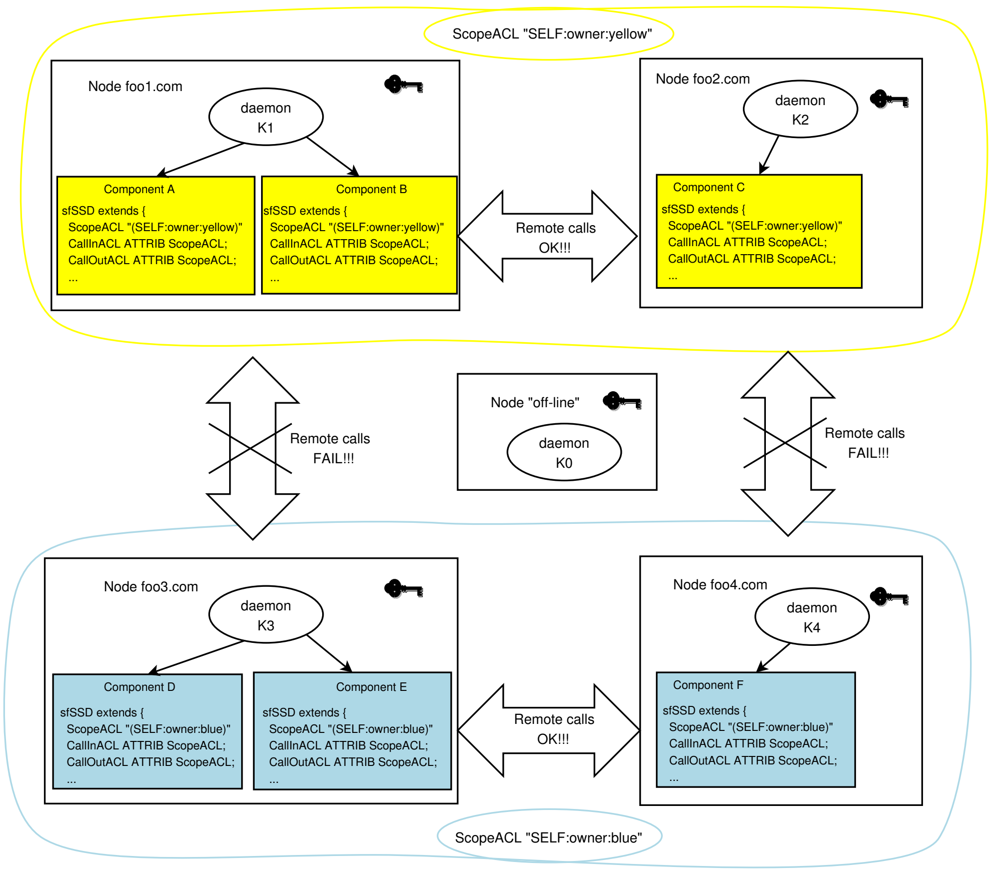

ScopeACL "SELF:owner:yellow"
Node foo1.com
daemon K1
Component A
sfSSD extends {
ScopeACL "(SELF:owner:yellow)"
CallInACL ATTRIB ScopeACL;
CallOutACL ATTRIB ScopeACL;
...
Component B
sfSSD extends {
ScopeACL "(SELF:owner:yellow)"
CallInACL ATTRIB ScopeACL;
CallOutACL ATTRIB ScopeACL;
...
Remote calls OK!!!
Node foo2.com
daemon K2
Component C
sfSSD extends {
ScopeACL "(SELF:owner:yellow)"
CallInACL ATTRIB ScopeACL;
CallOutACL ATTRIB ScopeACL;
...
Remote calls FAIL!!!
Node "off-line"
daemon K0
Remote calls FAIL!!!
Node foo3.com
daemon K3
Component D
sfSSD extends {
ScopeACL "(SELF:owner:blue)"
CallInACL ATTRIB ScopeACL;
CallOutACL ATTRIB ScopeACL;
...
Component E
sfSSD extends {
ScopeACL "(SELF:owner:blue)"
CallInACL ATTRIB ScopeACL;
CallOutACL ATTRIB ScopeACL;
...
Remote calls OK!!!
Node foo4.com
daemon K4
Component F
sfSSD extends {
ScopeACL "(SELF:owner:blue)"
CallInACL ATTRIB ScopeACL;
CallOutACL ATTRIB ScopeACL;
...
ScopeACL "SELF:owner:blue"

Figure 5: Using SmartFrog Security Domains.

than individual components. We will call a SmartFrog Security Domain (SSD) a group of components and daemons that enforce a consistent policy.

Figure 5 shows how we can create SSDs. Let's say that we want to create two SSDs ("yellow" and "blue") and associate components `A`, `B` and `C` to the first domain ("yellow") and components `D`, `E` and `F` to the second one ("blue"). Moreover, we want to make sure that components in one domain only interact with components of their own domain, even though these components could be deployed in multiple physical nodes. First, we have to configure the security credentials of our SmartFrog daemons so that they can represent the security role associated to one SSD. We could do that manually with Ant commands (as we will describe in Section 5) and establish a set of trust relationships between principals, as described in the graph in Figure 6. Looking at this graph we can see that we treat `K0`, a possibly "off-line" principal, as a common "owner" for all the other nodes, i.e., `K1`, `K2`, `K3` and `K4`. `K0` also associates the local binding `yellow` with `K1` and `K2`, and the local binding `blue` to `K3` and `K4`. If you remember our discussion in Section 3.1, this

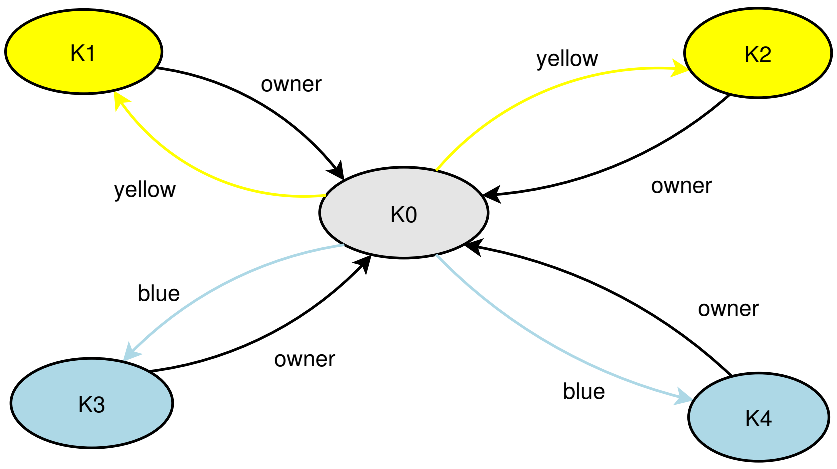


Figure 6: Configuring nodes credentials for example in Figure 5

means that `K1` and `K2` could prove the SPC `"(SELF:owner:yellow)"` to `K1`, `K2`, `K3` or `K4`; and we can make a similar argument for `K3` or `K4` with respect to the SPC `"(SELF:owner:blue)"`.

Now we are ready to properly describe the `ScopeACL` attribute of the `sfSSD`, that we introduced in Section 3.3. The purpose of this attribute is to describe with an SPC a requirement on the credentials of the daemon where this component should be deployed, so that we can later on use this requirement to define an SSD. In this way, we are implicitly associating this component to a daemon that belongs to that SSD. For example, component `A` in Figure 5 has a `ScopeACL` `"(SELF:owner:yellow)"` and is deployed in a node, `K1`, that can validate that SPC to the others. Therefore, we implicitly associate this component to the SSD defined by `"(SELF:owner:yellow)"`. However, if we try to deploy that component in node `K3`, that node should abort the deployment since it can only prove `"(SELF:owner:blue)"` and not `"(SELF:owner:yellow)"` to others. How can `K1` or `K3` know that?. They can only guess it, since they just rely on local information to do this check. So as long as it can prove it to "someone like me" using its locally cached credentials, i.e., the chain of certificates that was configured using Ant, they will assume that it is OK to continue the deployment, otherwise they will abort. So what if they get it wrong?. In most cases, the security of the system does not rely on the proper enforcement of `ScopeACL`, treating it more like a "hint" to detect configuration problems early on; instead, we use its value on "real" ACLs, like `CallInACL` or `CallOutACL`, to enforce the SSD. Note that support for "linking" and "templating" in the SmartFrog language is really useful to include the value of a `ScopeACL` in other ACLs, as we can see in following example (based on Figure 5):

```
yellowSSD extends LAZY SSD {
  ScopeACL "(SELF:owner:yellow)";
  CallInACL ATTRIB ScopeACL;
  CallOutACL ATTRIB ScopeACL;
}
blueSSD extends LAZY yellowSSD {
  ScopeACL "(SELF:owner:blue)";
}
// Yellow SSD for components A, B and C
componentA extends Prim {
  sfSSD ATTRIB yellowSSD;
  ...
}
...
// Blue SSD for components D, E and F
componentD extends Prim {
   sfSSD ATTRIB blueSSD;
   ...
}
...
```

It is critical that we choose well the SPCs that define the SSDs, i.e., the ones that we use in a `ScopeACL`. The following properties are useful when we want an SSD to abstract the security properties of a group of components and daemons:

Consistency all the relevant principals, i.e., principals that try to enforce an ACL based on a particular `ScopeACL` defining an SSD, agree on whether another principal satisfies or not the `ScopeACL`.

Equivalence membership in an SSD is defined using an equivalence relation based on whether principals satisfy a particular `ScopeACL`. This implies that this relation is reflexive, i.e., a principal can prove to itself `ScopeACL`, symmetric, i.e., two principals in the SSD always can prove it to each other, and transitive, i.e., assuming principals `K1`, `K2` and `K3` are in the SSD, if `K1` can prove it to `K2` and `K2` to `K3` then `K1` can prove it to `K3`.

Separability if a member of an SSD can prove the `ScopeACL` of another SSD to a relevant principal, then any principal that is member of the first SSD should be able to prove it too.

Partial-order whether members of an SSD can prove the `ScopeACL` of another SSD defines a partial order between SSDs. The motivation is to make easier to understand dependencies between domains. Clearly, we want the **Separability** and **Equivalence** properties to hold too, so we can extend the partial order to principals in the corresponding SSDs. As in the **Equivalence** case, this relation is reflexive and transitive; however, we would like it to be antisymmetric, i.e., if members of an SSD prove the `ScopeACL` of another domain, then members of that domain cannot prove the `ScopeACL` of the first SSD, unless we are really talking about a single SSD.

Granularity all ACLs that we use in the system are obtained by combining, i.e., with "OR" semantics, the `ScopeACL` of other SSDs. Therefore, the "finest" granularity in an access control decision is always defined by an `ScopeACL`.

Completeness all the relevant principals can be mapped to some SSD.

Clearly, all these properties are not independent from each other, and not all of them are always required to have a useful SSD-based decomposition. For example, if we satisfy **Consistency** for all principals and all scopes we can trivially achieve **Equivalence**, since we have a partition of principals into equivalence classes based on a "global" property. **Partial-order** is useful when we try to understand the security implications of a node compromise in other parts of the system, but we can always do a more "conservative" analysis if needed. **Granularity** and **Completeness** allow us to ignore any SPC that is not part of a `ScopeACL`, and this simplifies the analysis, but we could find useful to add well-defined SPCs, e.g., (`<PUBKEY>`), without adding new scopes. **Separability** allows us to generalise the access rights from one principal to the rest of the SSD, but in some scenarios we have a "designated" principal that interacts with members of another SSD, and we just worry about that principal. The recommendation is not to be too "dogmatic" about properties, and just use them when it makes sense, e.g., when we want to understand better the requirements of a particular design.

Let's look at how "compliant" are the `ScopeACLs` chosen for the example in Figure 5:

Consistency `K1` an `K2` enforce `"(SELF:owner:yellow)"` and they agree that only `K0`, `K1` and `K2` are OK; `K3` and `K4` enforce `"(SELF:owner:blue)"` and they agree that only `K0`, `K3` and `K4` are OK; `K1`, `K2`, `K3` and `K4` enforce `"(SELF:owner)"` and they agree that only `K0` is OK.

Equivalence `K1` and `K2` trivially satisfy a reflexive, symmetric and transitive relation based on satisfying `"(SELF:owner:yellow)"`. The same argument applies to `K3` and `K4` with `"(SELF:owner:blue)"`. `K0` is the only member of `"(SELF:owner)"`, so **Equivalence** is trivial in that case.

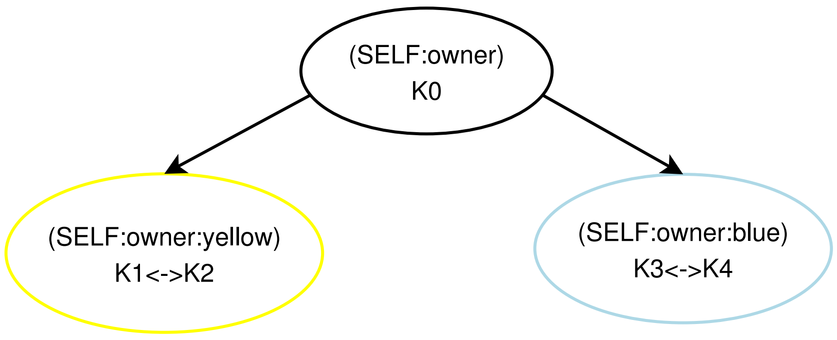


Figure 7: Partial-order between SSDs.

Separability `K0` is the only member that can prove other `ScopeACLs`, and it is only one principal; we are OK.

Partial-order "`(SELF:owner)`" implies "`(SELF:owner:yellow)`" and "`(SELF:owner:blue)`"; this relationship is antisymmetric and transitive.

Granularity all the ACLs only use the `ScopeACL` of other SSDs.

Completeness `K0` belongs to "owner", `K1` and `K2` to "yellow", and K3 and K4 to "blue". So all are covered by SSDs.

So our example satisfies all the properties of a "good" SSD-based decomposition. The benefit is that we just need to understand simple trust relationships between SSDs and forget about components and daemons (see Figure 7). We will show other more complex examples in this document that are also "compliant". However, when the relationships become very complex, automated tools that check for desired properties or come up with a "good" design can be very useful.

3.5 Interaction across SSDs

One of the critical motivations for the new security model was to allow safe interactions with less trusted peers. So far we have only described how to create SSDs and how to allow interactions among components within a SSD. In this section we generalise these interactions to members of different SSDs.

The simplest way to achieve this type of interactions is to open all the components of an SSD to members of another SSD. For instance, we could modify our example in Figure 5 so that members of the "yellow" SSD can call remote methods of components in the "blue" SSD as follows:

```
yellowSSD extends LAZY SSD {
  ScopeACL "(SELF:owner:yellow)";
  CallInACL ATTRIB ScopeACL;
  // allow call-out to blue
  CallOutACL ATTRIB ScopeACL ++ "OR" ++ ATTRIB blueSSD:ScopeACL;
 }
blueSSD extends LAZY yellowSSD {
  ScopeACL "(SELF:owner:blue)";
  // allow call-in from yellow
  CallInACL ATTRIB ScopeACL ++ "OR" ++ ATTRIB yellowSSD:ScopeACL;
  CallOutACL ATTRIB ScopeACL;
 }
// Yellow SSD for components A, B and C
componentA extends Prim {
   sfSSD ATTRIB yellowSSD;
  ...
}
...
// Blue SSD for components D, E and F
componentD extends Prim {
    sfSSD ATTRIB blueSSD;
    ...
}
...
```

Note the use of keyword `OR` to add the `ScopeACL` of another SPC to the enforced ACL. Also, note how this description is simplified by using built-in operators in the SmartFrog language for concatenating strings and linking attributes. Clearly, an RMI interaction across SSDs can only take place if both the outgoing calls in the "yellow" SSD and the incoming ones in the "blue" SSD are allowed.

Unfortunately, the previous approach has several weaknesses: a compromise of a principal in the "yellow" SSD could lead to a serious threat for *all* the principals in the "blue" SSD, since all their remote methods are available to it; if instead we have a compromised principal in the "blue" SSD, it could affect *any* relevant[8] member of the "yellow" SSD by returning "rogue" results to its RMI calls.

How can we minimise the risk of cross-SSD interaction? We need to have tighter control on what is being shared and by whom between domains. We achieve this by identifying a special component in the "privileged" SSD as the

[8]In this context, the only "relevant" principals in the "yellow" SSD are the ones that interact with principals in the "blue" SSD.

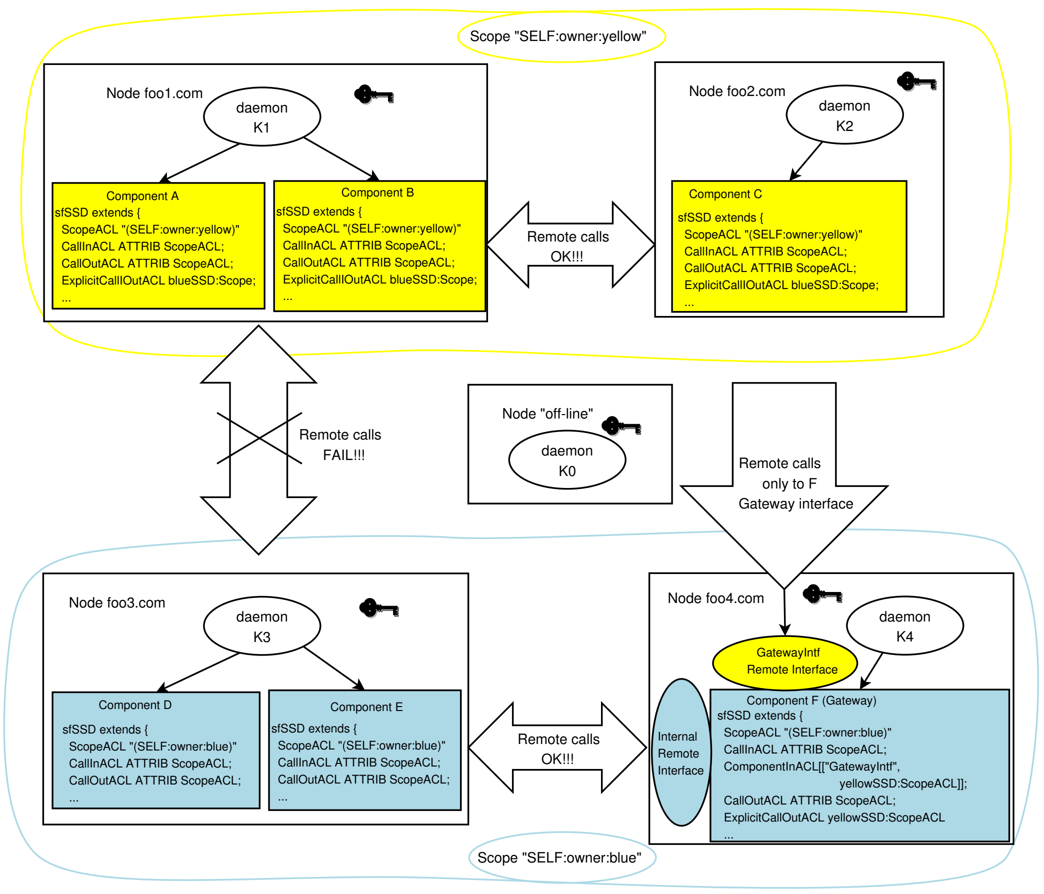
only accessible entry point for other SSDs, i.e., this component becomes a "gateway" that can be used by other SSDs to access components inside the "privileged" SSD. Moreover, "external" access is restricted to remote methods in a particular interface of the "gateway" component; so that by "hardening" or "limiting" the functionality of these methods, we can override safely the default security policy associated with the "privileged" SSD. Clearly, "internal" access should not be restricted, and the "gateway" component behaves as a "normal" component for other principals in the "privileged" SSD. Also, a "gateway" component sometimes makes outgoing RMI calls to components in "non-privileged" SSDs, and it is critical that it does not confuse remote references to "privileged" and "non-privileged" components, i.e., we need an explicit mechanism to allow outgoing calls to "non-privileged" components.

We include new optional attributes in `sfSSD` so that we can describe constraints of "gateway" components:

ComponentInACL its value is a vector of interface-SPC pairs. Each pair refines the `CallInACL` policy of the component by defining and extra SPC that can be used to invoke methods in that particular remote interface. Note that some remote methods could be described in multiple interfaces, and we should not try to define conflicting policies for these interfaces; otherwise, we will have a run-time exception during deployment.

ExplicitCallOutACL an extra SPC that refines the `CallOutACL` of this component for remote references that are explicitly "wrapped". The changes to the component implementation required to "wrap" a remote reference are described in Section 4.3. However, many core SmartFrog components already do this "wrapping" for you, as we will describe in Section 4.1.

Figure 8 shows how to apply these ideas to the previous example. We choose component `F` as our entry point in the "blue" SSD, i.e., a "gateway" component that can interact with components of the "yellow" SSD. We assume that all the "hardened" remote methods of `F` that can be called by components from the "yellow" SSD are in the interface `GatewayIntf`; all the other remote methods of `F` are accessible through some other "internal" interfaces. We use the `ComponentInACL` attribute in `sfSSD` to allow members of the "yellow" SSD to access `F`'s "hardened" methods. We also use the `ExplicitCallOutACL` so that `F` can call remote methods of "yellow" components after explicit "wrapping" of their remote references, i.e., their "stubs". Note that both `ComponentInACL` and `ExplicitCallOutACL` extend rather than replace the policy of `CallInACL` and `CallOutACL`, respectively; this means that they do not restrict interactions of `F` with other components in the "blue" SSD. Let's look at the description changes:



Component A
```
sfSSD extends {
  ScopeACL "(SELF:owner:yellow)"
  CallInACL ATTRIB ScopeACL;
  CallOutACL ATTRIB ScopeACL;
  ExplicitCallOutACL blueSSD:Scope;
  ...
```

Component B
```
sfSSD extends {
  ScopeACL "(SELF:owner:yellow)"
  CallInACL ATTRIB ScopeACL;
  CallOutACL ATTRIB ScopeACL;
  ExplicitCallOutACL blueSSD:Scope;
  ...
```

Component C
```
sfSSD extends {
  ScopeACL "(SELF:owner:yellow)"
  CallInACL ATTRIB ScopeACL;
  CallOutACL ATTRIB ScopeACL;
  ExplicitCallOutACL blueSSD:Scope;
  ...
```

Component D
```
sfSSD extends {
  ScopeACL "(SELF:owner:blue)"
  CallInACL ATTRIB ScopeACL;
  CallOutACL ATTRIB ScopeACL;
  ...
```

Component E
```
sfSSD extends {
  ScopeACL "(SELF:owner:blue)"
  CallInACL ATTRIB ScopeACL;
  CallOutACL ATTRIB ScopeACL;
  ...
```

Component F (Gateway)
```
sfSSD extends {
  ScopeACL "(SELF:owner:blue)"
  CallInACL ATTRIB ScopeACL;
  ComponentInACL[["GatewayIntf",
                  yellowSSD:ScopeACL]];
  CallOutACL ATTRIB ScopeACL;
  ExplicitCallOutACL yellowSSD:ScopeACL
  ...
```

Figure 8: An example of a cross-SSD “gateway”.

```
commonSSD extends LAZY SSD {
   CallInACL ATTRIB ScopeACL;
   CallOutACL ATTRIB ScopeACL;
}
yellowSSD extends LAZY commonSSD {
  ScopeACL "(SELF:owner:yellow)";
  ExplicitCallOutACL ATTRIB blueSSD:ScopeACL;
}
blueSSD extends LAZY commonSSD {
  ScopeACL "(SELF:owner:blue)";
}
gatewaySSD extends blueSSD {
  ComponentInACL [["org.foo.GatewayIntf",
                   ATTRIB yellowSSD:ScopeACL]];
  ExplicitCallOutACL ATTRIB yellowSSD:ScopeACL;
}
// Yellow SSD for components A, B and C
componentA extends Prim {
  sfSSD ATTRIB yellowSSD;
  ...
}
...
// Blue SSD for components D and E
componentD extends Prim {
  sfSSD ATTRIB blueSSD;
  ...
}
...
// Gateway SSD for component F
 componentF extends GatewayFoo {
   sfSSD ATTRIB gatewaySSD;
   ...
 }
```

We have a way to relax the security policy of a remote interface but how about "hardening" the implementation of the exposed methods in that interface? In some cases we do not need to change their implementation because we are just dealing with "safe" methods, e.g., a "read-only" method; most methods in the `ProcessCompoundACL` interface, which is part of the SmartFrog core, are of that nature. In other cases we just need to do extra validation of the arguments or "wrap" remote references so that the `ExplicitCallOutACL` works as intended.

The most complex scenario is when we want to change the behaviour of a method depending on whether the request came from the "privileged" SSD or the "non-privilege" one; the `GatewayImpl`, also in the SmartFrog core, implements that behaviour to achieve a "chroot" environment for a sub-tree of components. We will look in some detail at the `GatewayImpl` example in Section 5. Also, how to access programmatically security information associated with an incoming call is described in Section 4.3.

4 Enforcing the Security Model

After describing security relationships between distributed components, we need a way to enforce them. First, we describe the mechanisms used during component deployment that authenticate and interpret these descriptions. Then, we focus on resource loading and how it interacts with security. Finally, we look at "on-line" remote interactions between components based on RMI and how their secure channels can be customised by security attributes.

The main purposes of this section are: clarify the limitations of our implementation, illustrate the use of the security APIs (Application Programming Interface), and allow a better understanding of the examples. However, its main goal is not to document in detail the internals of our implementation, i.e., the discussion is kept at a "high-level".

4.1 Deploying signed descriptions

One of the most important novelties introduced with the new security is to be able to sign descriptions, and make deployment decisions based on these signatures and not only on the credentials of the cooperating peers. This allows peers that do not fully trust each other to safely coordinate actions in a common deployment task. Figure 9 gives us an overview of the deployment process, highlighting the security checks performed:

1. In order to parse a description we load in the JVM the description file, i.e., main file with extension .sf, and all the included sub-descriptions, i.e., referred by other descriptions that use the `#include` directive. All these files should be inside signed jar files[9] and, at load time, we will check that they

[9]Do not confuse the signature of the jar file that contains the .sf (and .class) files with the recursive signature that we compute after parsing the description. The first one does not "understand" the structure of the description, and it is used by the deployer to guarantee the authenticity of "templates" and other resources needed to create "deployment instructions" at parse time. The second one is computed recursively by the deployer to authenticate the actual "deployment instructions", and it is relevant during deployment time.

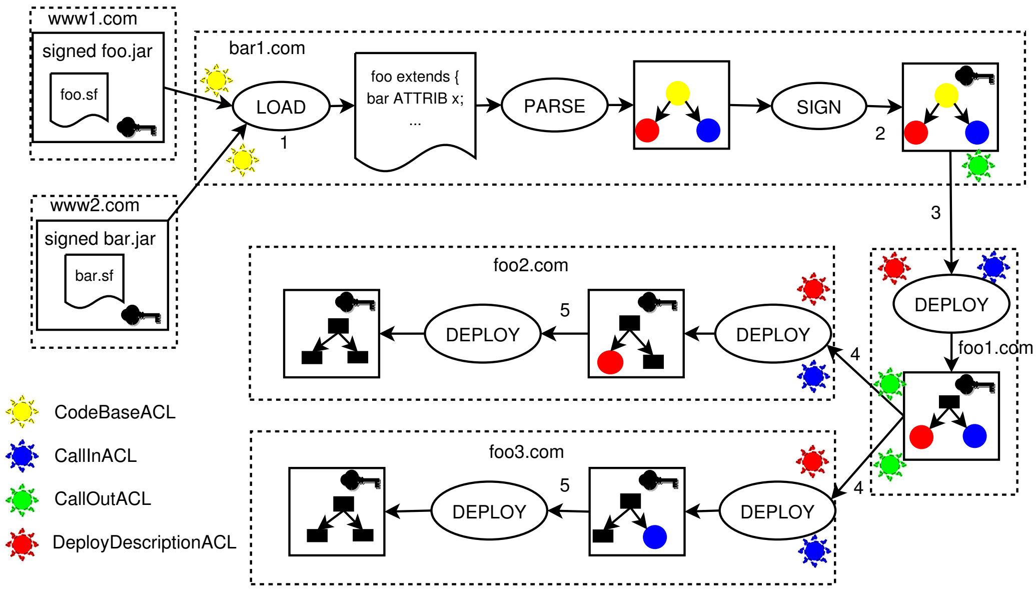


Figure 9: Signing and deploying component descriptions.

satisfy `CodeBaseACL`. Note that these jar files could be remotely loaded from a web server by setting the appropriate codebase property (see Section 4.2).

2. After we execute all the parser phases, we obtain a completely processed description that becomes the input of the signing operation. This operation annotates the data structure with a recursive signature that authenticates the description. We describe later on this operation in detail.

3. We deploy this signed description in a remote node by performing an RMI call that propagates this description to the deployment target. In this case we have to satisfy our `CallOutACL` requirements and the target's `CallInACL` requirements in order to continue with the RMI call. Moreover, the target performs an extra check that ensures the description is properly signed and the signer credentials satisfy its `DeployDescriptionACL` policy.

4. The target starts deploying the description locally and finds that some sub-descriptions need to be forwarded to other nodes. It "collapses" unnecessary sub-descriptions, i.e., by replacing them by its hash value, before repeating step 3 for each sub-target. Note that the target does not re-sign the sub-descriptions and the "collapsing" process does not affect the original signature, i.e., the sub-targets can verify the original signature from a partial description. This means that even if the credentials of the first target cannot satisfy the `DeployDescriptionACL` of the sub-targets, i.e., it

cannot deploy new descriptions, it can successfully propagate the original "deployment instructions".

5. The sub-targets finish the local deployment and enforce the security policies associated with the new components. This means that new security relationships between nodes foo1, foo2 and foo3 could have been established as the result of this deployment. For example, the deployed components could enforce a new DeployDescriptionACL that allows foo1 to sign and deploy arbitrary descriptions.

A bootstrapping problem of the previous approach is how to guarantee that CallInACL and CallOutACL policies between partially trusted peers are satisfied. In SmartFrog, a typical deployment bootstraps by using the root ProcessCompound component, i.e., the first component deployed when the daemon starts, and it becomes a "local parent" for components that do not have a "real parent". Then, it is a natural approach to also use this special component to bootstrap security: we solve the CallInACL bootstrapping problem by making the ProcessCompound a "gateway" component (see Section 3.5), with a "non-privilege" interface ProcessCompoundACL that allows other partially trusted peers[10] to invoke safe component look-up and remote deployment methods, i.e., methods that always validate signed description arguments or are "read-only". We solve the CallOutACL bootstrapping problem by transparently "wrapping" with a permissive ExplicitCallOutACL the remote ProcessCompound components that we find with the RootLocator object. These policies can be changed by modifying processcompound.sf, but the default sfSSD that implements them is:

```
ProcessCompound extends Compound {
  sfSSD extends LAZY SSD {
    ComponentInACL[
      ["org.smartfrog.sfcore.processcompound.ProcessCompoundAcl",
       "(SELF:...)"]];
    ExplicitCallOutACL "(SELF:...)";
  }
}
```

Let's explain how descriptions are signed and checked (see [4] for a more detailed description). We always sign a canonical representation of a fully resolved SmartFrog description; the idea is that small changes to a description that do not affect its "meaning", e.g., spacing, should not invalidate a signature. We define the canonicalisation process by induction since after full resolution we just have

[10]By default partially-trusted peers should validate the SPC "(SELF:...)".

a simple hierarchy that aggregates basic types: we specify one transformation for the basic types and another one for a group of already transformed types. This canonical transformation should have the following properties:

Validity its output is a valid SmartFrog description that can be parsed, deployed,...

Fixed point if we apply this transformation again, i.e., using the previous output as its new input, we just obtain the same output again.

Weak equality if we deploy its canonical output instead of the original description, we should obtain exactly the same deployed components. This implies that we can know if two descriptions are "equal", i.e., they will produce the same deployed components, by obtaining their corresponding canonicalised versions and doing a string comparison operation[11] with them. Note that this only gives us a "weak" definition of equality since we could have different canonical descriptions that produce indistinguishable deployments, e.g., some attributes are ignored at deployment time, but this does not affect the security properties that we guarantee.

Robust parsing let's say that we parse the canonical output to obtain an object representation and then unparse this representation to obtain a textual description again. If we apply the canonicalisation to this textual description we should obtain the same result, i.e., parse/unparse operations do not affect canonicalisation.

If we want to authenticate a complete description, we could just use a "conventional" signature scheme that computes a hash, i.e., SHA-1, over a binary encoding of the canonical output of the previous transformation. Figure 10 shows the steps that we could use to secure deployment of complete descriptions with that approach:

1. Transform the object representation of the description into textual form.
2. Apply the canonicalisation process to obtain a canonical textual form.
3. Compute a hash of the binary encoding of the canonical textual form.
4. Check the signature attached to the object representation using this hash.
5. Check that credentials associated with the signature satisfy our `DeployDescriptionACL` policy.

[11]We fix the character encoding format so this is equivalent to equality of byte arrays.

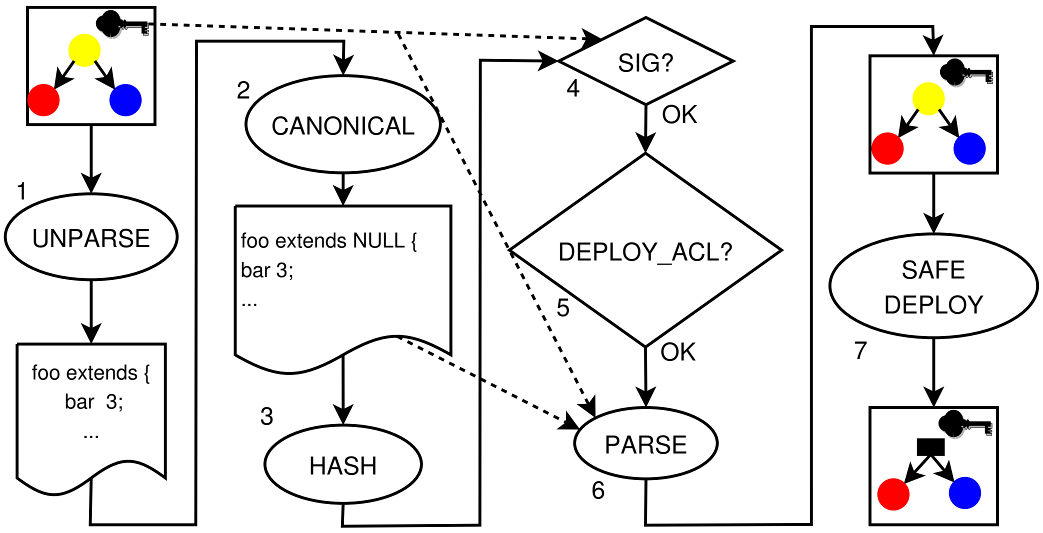


Figure 10: Description deployment with conventional signatures.

6. Parse the canonical textual form to obtain an equivalent object representation; attach to this representation the original signature.

7. Continue with the deployment using the new object representation.

Note that the input object representation of the description is treated as "tainted" data, and we need to unparse/parse it so that it can be trusted. In practise, our implementation uses a "safe cloning" strategy to achieve the same goal in a more efficient manner.

However, if we look at the deployment process in Figure 9, we do not want to propagate a full description all the time, we just need the bits that are relevant for the current deployment sub-tree. Also, as we will see in Section 6.1, sometimes we want to allow late customisation of certain parts of a description in a controlled way. All these requirements need a different way of hashing the canonical representation, i.e., a hashing function that understands the structure of the description.

The recursive hashing scheme that we have implemented is very similar in nature to the canonicalisation process. For each basic element we compute the hash directly from its canonical format. For each aggregate element, we first replace the value of each (non-basic) children by a string representation of their hash; then, we just digest the resulting canonical string. We do this operation in a post-order traversal of the hierarchy, so we guarantee that we have computed the children hashes first. For example, if the initial canonical input is:

```
{
```

```
foo extends LAZY {
bar 3;
otherBar extends LAZY {
bar 7;
};
};
}
```

and let's assume that we have a function `hashString(String)` that will return an string representation of the SHA-1 digest of an input string, i.e., we assume a known binary character encoding. So in the post-order traversal of the hierarchy we first apply `hashString` to the string value of `otherBar`, i.e., `"extends LAZY {\nbar 7;\n}"`, obtaining `"sdsdasdasd"`, and we get:

```
{
foo extends LAZY {
bar 3;
otherBar HASH-SHA-1#sdsdasdasd#;
};
}
```

then we apply `hashString` to the new string value of `foo`, i.e., `"extends LAZY {\nbar 3;\n otherBar HASH-SHA-1#sdsdasdasd#;\n}"` to obtain `"errwerwsd"`, and we get:

```
{
foo HASH-SHA-1#errwerwsd#;
}
```

and finally we can directly compute the hash of `"{\nfoo HASH-SHA-1#errwerwsd#;\n}"` by using `hashString`.

Figure 11 shows how we actually implement secure deployment using a recursive hashing scheme. The main difference with the process in Figure 10 is that we safely clone the object representation and then use this clone to drive both the recursive computation of the hash and, if needed, the local deployment. This means that the canonical textual representation is implicit in the cloned object representation.

To support late customisation we can stop the recursive hashing process by marking the top component of a sub-description with the keyword `SIGNED`; then, this sub-description will have its own signature, and changes to it will not affect the top level signature. However, we specify in the enclosing description who can

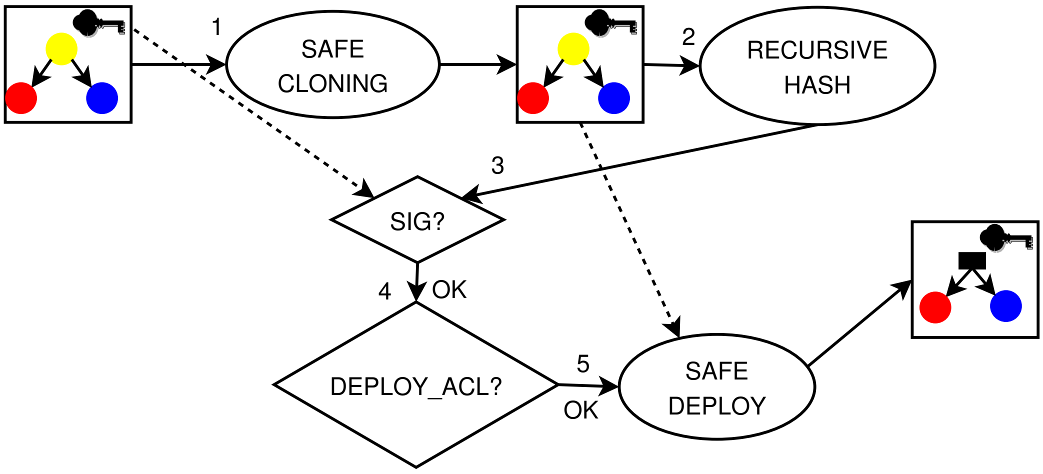


Figure 11: Description deployment with recursive hashing.

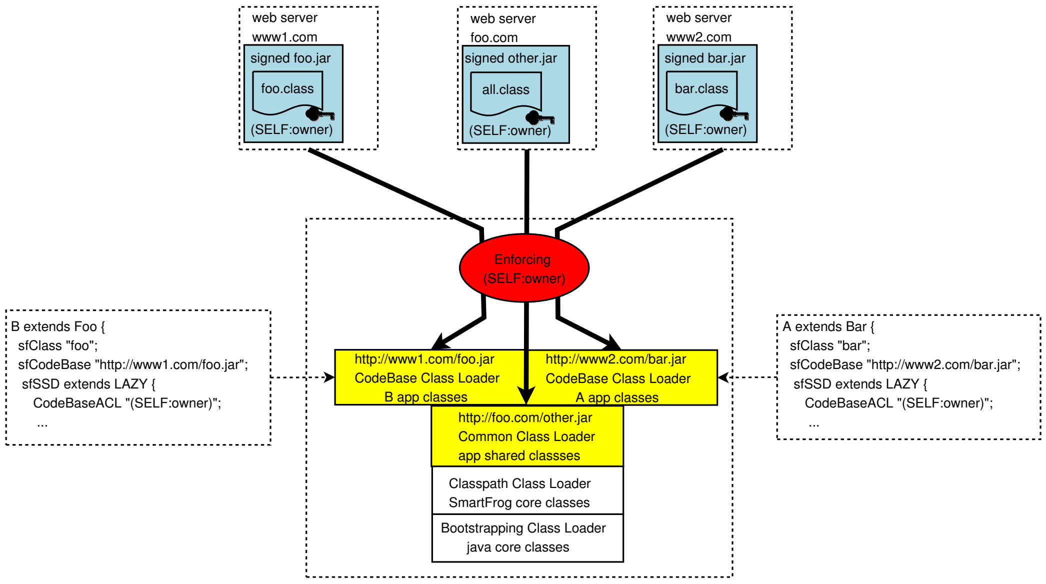
actually sign this sub-description, i.e., the top level signature *will* change if we try to add another valid signer. More on signing sub-descriptions in Section 6.1.

So how do we create signed descriptions in practise? The easiest way is to parse a ".sf"file using the default parser settings, i.e., assuming the security mechanisms have been initialised, and the resulting ComponentDescription will have attached a signature that include your public security credentials:

```
InputStream in =
  SFClassLoader.getResourceAsStream("http://foo.com/bar.jar",
                                    "foo.sf");
SFParser parser = new SFParser("sf");
// "conventional" parsing process
Phases phases = parser.sfParse(in).sfResolvePhases();
// "transparent" description signing in here...
ComponentDescription cd =  phases.sfAsComponentDescription();
SFSignature sig = cd.getSignature();
```

However, in some cases we have programmatically created the ComponentDescription, or we have changed it later, i.e., invalidating a previous signature. In those cases we can add a signature (or replace a previous one) to a top level ComponentDescription object, i.e., one with a null "parent", as follows:

```
ComponentDescription topcd = .... // whatever constructor
SFDescriptionSigner.signDescription(topcd);
SFSignature sig = topcd.getSignature();
```



```
B extends Foo {
  sfClass "foo";
  sfCodeBase "http://www1.com/foo.jar";
   sfSSD extends LAZY {
     CodeBaseACL "(SELF:owner)";
     ...
```

```
A extends Bar {
  sfClass "bar";
  sfCodeBase "http://www2.com/bar.jar";
   sfSSD extends LAZY {
     CodeBaseACL "(SELF:owner)";
     ...
```

Figure 12: Overview of SmartFrog class loading.

4.2 Dynamic class loading

In SmartFrog we use dynamically created class loaders to load resources from remote sources, i.e., web servers. These sources are typically specified within the deployment description, i.e., using the `sfCodeBase` attribute, but the authentication of resources relies on signatures associated with the jar files that contain them, and not on the sources themselves. In the new implementation we extend this mechanism so that we can enforce a particular `CodeBaseACL` on the signer of the jar file. This ACL is also specified in the description, as we described in Section 3.3. Other new features of this implementation include: allowing to reset class loading without stopping the SmartFrog daemon, supporting dynamic changes of the common codebase property, and a better integration with the RMI class loading mechanisms that download stubs dynamically.

In order to implement these changes we had to change slightly how we manage secure class loaders. We now rely on our own class loader implementation to enforce the `CodeBaseACL` checks, and not on the use of a particular resource loading API. This means that after creating a “special” class loader, it can be treated as any other “standard” java class loader. Also, we do not rely on the default RMI implementation of class loaders caching; instead, we have our own caching implementation and we configure RMI to use it; therefore, it is easier to reset class loading, i.e., clean the cache of class loaders, and it forces RMI to use our “special” class loaders to load stubs.

Figure 12 shows a typical stack of class loaders during secure component deployment. We identify four different types of class loaders:

Bootstrapping a "standard" class loader that java uses to load system classes.

Classpath a "standard" class loader that java uses to load "local" application classes from sources specified in the `CLASSPATH` environment variable (or using a command line option, i.e., `-classpath`). We use it to load smartfrog core classes. Security relies on standard java 2 mechanisms [1], i.e., a java policy file that grants all permissions to classes in a jar file signed by the key "owner", i.e., the default key label used in our build process (see Section 5).

Common a "special" class loader that enforces a `CodeBaseACL` for remotely loaded classes and resources common to all components. The URLs that specify source jar files are obtained from the property `org.smartfrog.codebase`. If this property is not set we just default to the `Classpath` class loader. All loaded resources should be inside signed jar files.

CodeBase a "special" class loader created "on-demand" that enforces a `CodeBaseACL` for loading remote classes and resources from a codebase URLs specified in a deployment description. All loaded resources should also be inside signed jar files.

These class loaders form a "normal" class loader hierarchy, with the `BootStrapping` class loader at the top, then a `Classpath` class loader, next the `Common` class loader, and the `CodeBase` class loader at the bottom . For example, if we ask a `CodeBase` class loader to load a class, we first check if the class is available in the `Bootstrapper` class loader, then in the `Classpath` class loader, later in the `Common` class loader, and finally we use the `CodeBase` class loader. This order ensures that we re-use classes as much as possible.

As we just mentioned, when RMI dynamically loads stub classes from the annotated URLs it also uses `CodeBase` class loaders, since we replace the default RMI class loader factory. Our management of class loaders is very similar to the default RMI implementation, and we cache class loaders from a particular URL until we do a reset. A simple way to obtain a `CodeBase` or `Common` class loader, which can be used as a "conventional" class loader, is to use the standard RMI class loader API:

```
ClassLoader cl;
// cl is a "normal" class loader that imposes CodeBaseACL
cl = RMIClassLoader.getClassLoader("http://foo.com/bar.jar");
// cl is the "common" class loader when codebase is null
cl = RMIClassLoader.getClassLoader(null);
```

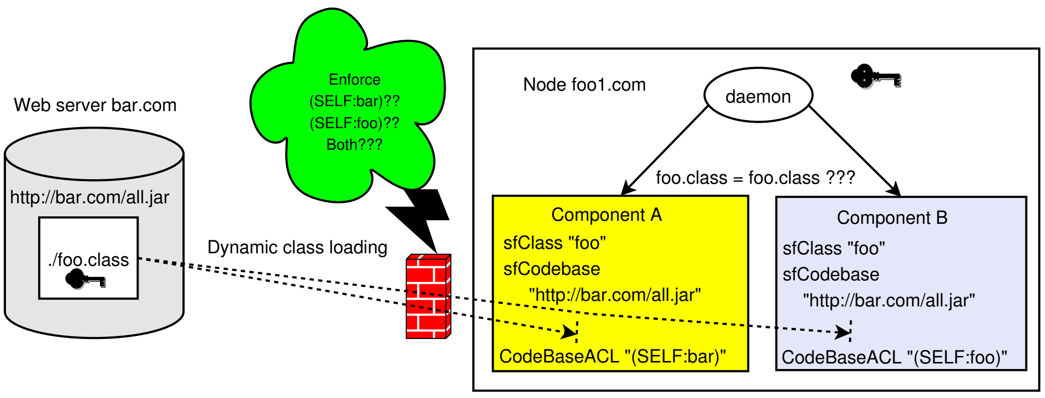


Figure 13: `CodeBaseACL` policy conflict.

However, we still support the "old" resource loading API that uses static methods in the class `SFClassLoader`. This class is now implemented as a wrapper that creates the "special" class loader, if needed, and delegates the action to it.

Figure 12 also suggests that a common `CodeBaseACL` is enforced by all the cached `CodeBaseACL` class loaders and the current `Common` class loader. The ACL that a class loader enforces is frozen at creation time; therefore, every time that we modify this common `CodeBaseACL` by deploying a component with a different policy, we flush the cache of class loaders, so that a "fresh" class loader will be created for the new policy even if we ask for an "old" codebase. This is implemented by triggering a reset of the class loading, that also re-reads the `org.smartfrog.codebase` property and creates a "fresh" `Common` class loader with the new codebase and ACL[12]. Note that we do not need to change `CodeBaseACL` to trigger this reset, we could do it more directly by using:

```
    SFClassLoader.resetClassLoading();
```

Unfortunately, resetting class loading due to a conflict of policies can have unwanted consequences. Figure 13 shows what could happen when we specify a different `CodeBaseACL` for two locally deployed components. In this case we dynamically load the class `foo.class` from an external web server in order to deploy components `A` and `B`. Even though the class name and codebase are identical for both components, the ACL is not, and we will trigger a class loading reset before deploying the second component; i.e., the same class is loaded twice, using

[12]Note that the `Bootstrapping` and `Classpath` class loaders never change. Therefore, we cannot change the SmartFrog core classes without stopping the daemon.

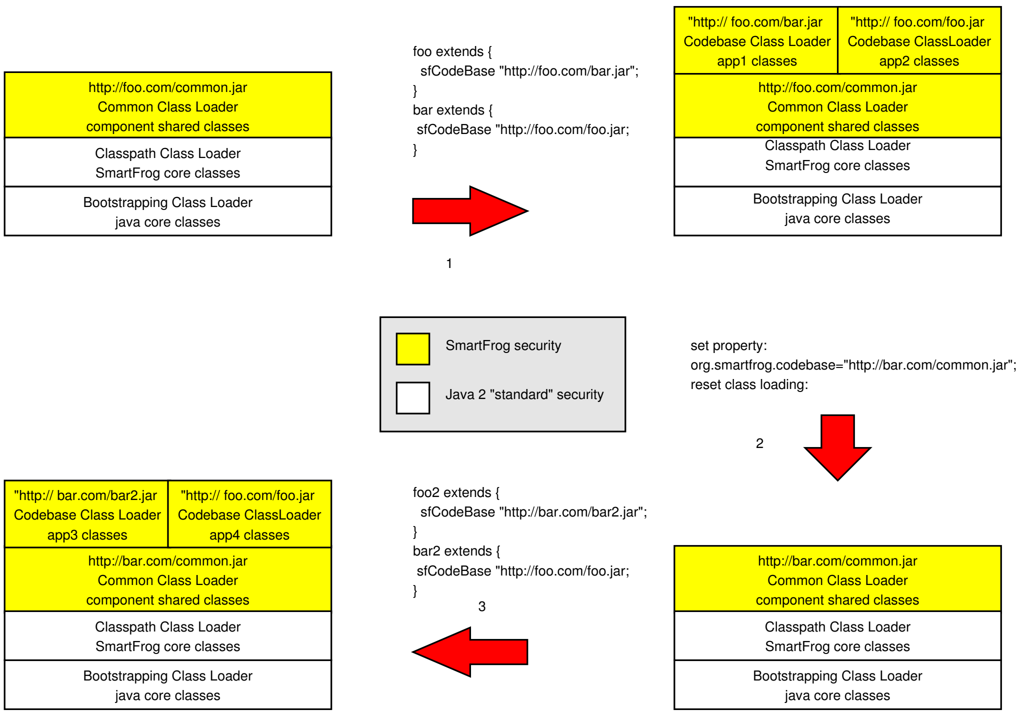


Figure 14: Managing dynamic class loaders

a different class loader to enforce each policy. This means that components A and B are instances of different classes and you are likely to get unexpected class cast exceptions during assignment.

Moreover, an application could chose to re-use its own class loaders, instead of relying on SmartFrog's caching mechanisms. Since resetting class loading will not affect existing class loaders, we could be loading classes with them that use "old" security policies. However, this is important to applications that want to isolate themselves from class loading configuration changes...

Dynamic class loading with changing security policies can be confusing, we truly recommend to limit policy changes to full resets, i.e., when we also terminate existing components. This will ensure that there are no unexpected interactions with previously loaded classes and a consistent security policy is enforced for all the loaded resources.

To summarise, Figure 14 shows an example of how the cache of class loaders changes when we deploy new components or force a reset:

- The starting point is a "fresh" SmartFrog daemon that has loaded the core classes from the `Classpath` and `Bootstrapping` class loaders. It also has installed a `Common` class loader for future component classes.

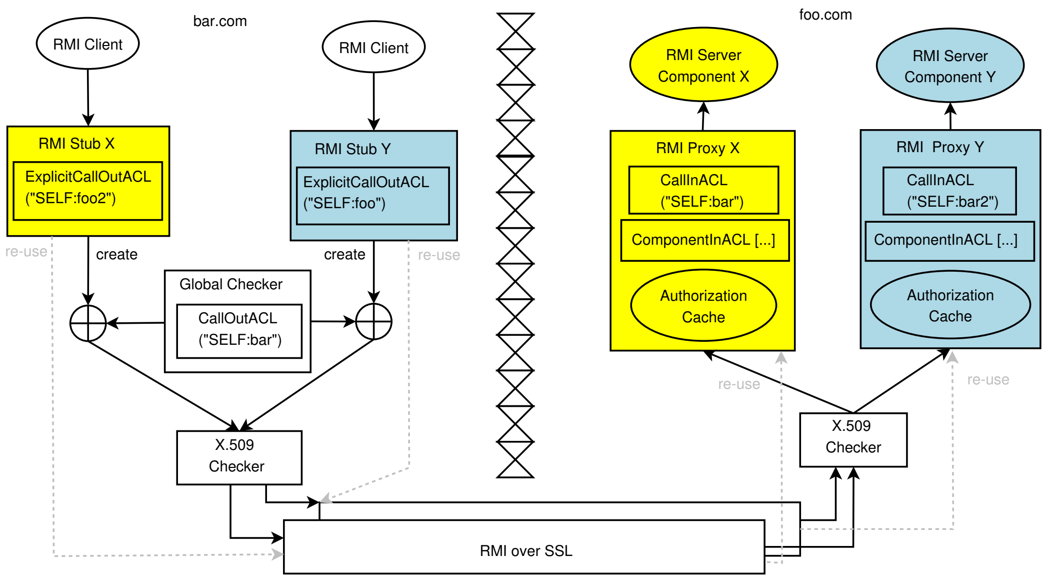


Figure 15: Secure RMI implementation.

- Deployment of two components with different codebases, but similar `CodeBaseACL`, triggers the creation and caching of two `CodeBase` class loaders.

- First, we change the codebase java property `org.smartfrog.codebase` to "`http://bar.com/common.jar`". Then, we force a class loading reset by either deploying a new component with a different `CodeBaseACL` or by using directly the SmartFrog APIs. After the reset, we have a new `Common` class loader that uses the new codebase and `CodeBaseACL`, and an empty cache of `CodeBase` class loaders.

- We deploy two new components with explicit codebases in their descriptions. Even though one of the codebases matches a previous one, i.e., it was used in a component deployed before the reset, we create and cache new class loaders for both.

4.3 Secure RMI (Remote Method Invocation)

In SmartFrog, when security is activated, every RMI interaction between components is tunnelled through secure channels, i.e., a mutually authenticated SSL socket. In Section 3.3 we described how to impose extra requirements on the principal at the other end of the channel by specifying attributes `CallOutACL` and `CallInACL` in the component's description. In Section 3.5, we introduced two other `sfSSD's` inner attributes: `ComponentInACL` and `ExplicitCallOutACL`; these attributes are critical to implement "gateways" between SSDs, since they can extend a default `CallInACL` and `CallOutACL` for a particular component or interface.

Figure 15 gives us an overview of how all these ACLs are implemented. It layers on top of X.509 certificate chain verification and SSL support provided by standard java libraries; this is possible since our certificates form valid X.509 certificate chains (see Section 3.1). Then, the design becomes very asymmetric, with the server making authorization decisions based on the component instance and method invoked pair, and the client using the combination of a remote reference, i.e., a stub, and the current global policy for `CallOutACL`. Ideally, a client will allow an RMI call to continue if the other principal either validates the `ExplicitCallOutACL` associated with the stub or the current `CallOutACL`. Similarly, a server will only allow an incoming RMI call if either the method is in an interface that `ComponentInACL` associates with an SPC that the other end validates, or the caller satisfies the `CallInACL` in the target component's description.

A significant effort was made to reduce the overhead of the security mechanisms: when the client establishes a secure channel, it will be re-used without furher checks as long as RMI does not discard the connection[13]; similarly, the server keeps a cache per component instance with authorization decisions, and also RMI in the server side re-uses connections that save expensive SSL sessions. Unfortunately, since the RMI stack is in control of connection management, it is difficult to guarantee in all cases that a change of policy will have immediate effects. Also, local calls are not checked, and sequences of local and non-local calls could create unintended conflicts of policy. Let's illustrate these issues with some examples:

Figure 16 shows that, if we deploy two components `A` and `B` in the same SmartFrog daemon, and they have a different `CallOutACL, A` could make a local call to `B` and, as a side-effect of that call, `B` makes a remote call to other external object `C`. In this case it is not clear which `CallOutACL` policy to apply to this remote invocation (`A`'s, `B`'s or both?). We could have associated the `CallOutACL` to the remote reference - and not to the caller's context -, as we do with the `ExplicitCallOutACL`, but this creates other problems: do we have to explicitly wrap all stubs before using them?; does stub serialization change policy?; If a method call returns an object that implements a remote interface, do I have to check its policy before using it?; and so on... At the end we decided to keep it simple, and just maintain the current global policy for the `CallOutACL`, i.e., updated every time we deploy a component with a different `CallOutACL`. For example, in

[13]Note that a client does not reuse connections among callers that use stubs with a different `ExplicitCallOut`. That's why the example in Figure 15 needs two connections...

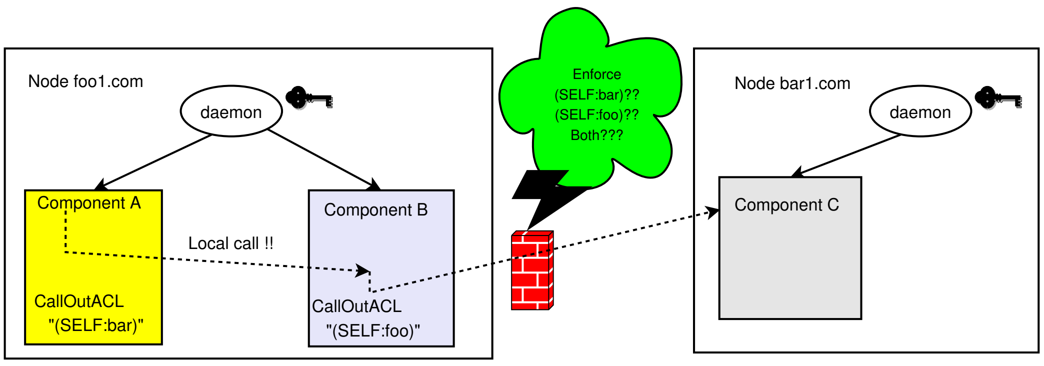


Figure 16: CallOutACL policy conflict

Figure 16, if component `B` was deployed after `A`, we will enforce `B`'s `CallOutACL` for all new outbound connections. Unfortunately, RMI's connection re-use could make very difficult to know if a new connection is needed for a particular call; so we just don't know whether `A`'s or `B`'s policy will be enforced, but *eventually* `B`'s policy will be the one enforced...

What if you really need to ensure that a particular outgoing RMI call will be satisfied if an ACL is satisfied? In that case you can use the `ExplicitCallOutACL`, i.e., change the java code so that it "wraps" the stub of the remote object before invocation, adding the `ExplicitCallOutACL` attribute to it. For example:

```
Prim foo = ... // local object that defines ExplicitCallOutACL
Prim remoteStubC = ... // A stub of the remote object
// wrap remoteStubC with the ExplicitCallOutACL defined by foo
remoteStubC = SFProcess.relaxCallOut(foo.getSecurityDomain(),
                                     remoteStubC);
// outgoing remote invocation check OK if:
//   -Remote object satisfies foo's ExplicitCallOutACL OR
//   -Remote object satisfies current default CallOutACL
remoteStubC.doit();
```

Note that for security reasons serialization of the stub will remove the `Explicit-CallOutACL`, i.e., we do not want a remote node affecting our local policies. Also, the RMI call continues if either the `ExplicitCallOutACL` **or** the current `CallOutACL` succeeds. This means we can only relax the requirements for doing a call out but not increase them, i.e., our "default" policy is always the most strict one; otherwise, the security of the system would rely on a very fragile assumption, that we are never using directly the un-wrapped stub - or a deserialized copy of it -. Our recommended approach is to start with a tight system and make explicit

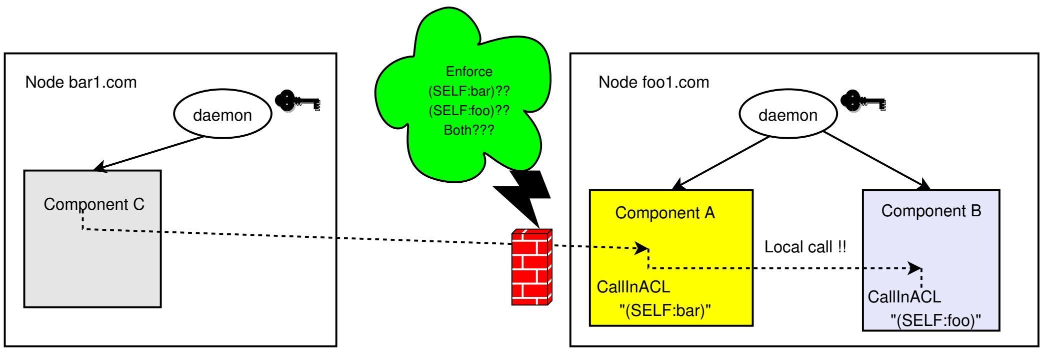


Figure 17: CallInACL policy conflict

actions to "open it up", rather than the other way round; however, this principle does not make us popular with the "usability" crowd...

Also, we can obtain more information about the principal at the other end *after* the invocation, since the context of the last successful remote call is associated with the currently executing thread:

```
//after the stub preparation of the previous example
remoteStubC.doit();
X509Certificate[] certs = null;
certs = SFNetworkUtil.getPeerAuthenticatedCallOutCerts();
specialLog(certs, remoteStubC, "doit");
```

Figure 17 shows a possible policy conflict scenario for incoming method call requests. In this case, component `C` makes a remote call to a method in component `A`, and that method invocation uses a local invocation to component `B`. If `C` had tried to call the same method in `B` directly, i.e., we assume it is also a remote method, he will need to satisfy the SPC `"(SELF:foo)"`; but instead, he is indirectly calling `B` via `A` and satisfying only `"(SELF:bar)"`. Our implementation in this case is always consistent, since we only enforce implicit checks in remote calls and `CallInACL` is never a global policy, i.e., only `A`'s policy is enforced. Moreover, as we saw in Section 3.5, this behaviour is useful to implement specially designed "gateway" components.

However, in some special cases we would like to know the credentials of someone calling a method remotely, even if the call is mediated by local calls. SmartFrog security also associates the credentials of the requesting principal to the executing thread, so that application code can query them for further security checks at any time, e.g.:

```
void foo() {
```

```
X509Certificate[] certs = null;
certs = SFNetworkUtil.getPeerAuthenticatedCallInCerts();
myCustomCheck(certs);
// Did the caller validate CallInACL or ComponentInACL?
boolean callInACLOK =
     SFNetworkUtil.isAuthenticatedDefaultCallIn();
...
```

5 Getting Started with the new SmartFrog Security

5.1 Initialisation

The basic set-up of SmartFrog security using Ant is, on the surface, very similar to what we had in the previous release. However, all the cryptographic operations are now performed by java classes, i.e., we no longer use openSSL to issue certificates, and we should be able to initialise a “CA” in Windows or Linux platforms without any extra software. For instance, we can create a "CA" from the `dist` SmartFrog directory with:

```
ant initCA
```

and this command will create security credentials for that principal, i.e., an encrypted keystore file and a properties file with its password, in the directory `dist/private/CA`. Now we can sign jar files with these credentials, and copy them to the directory `dist/signedLib` with the command:

```
ant signJars
```

Moreover, we can override the default jar files to be signed by changing the java property `libsToSign`, e.g.:

```
ant -DlibsToSign="SFGuiTools.jar" signJars
```

or by overriding the Ant target `getLibsToSign` in the `build.xml` configuration file. We can also delete the “CA” credentials with:

```
ant cleanCA
```

and create security credentials for a new peer with:

```
ant newDaemon
```

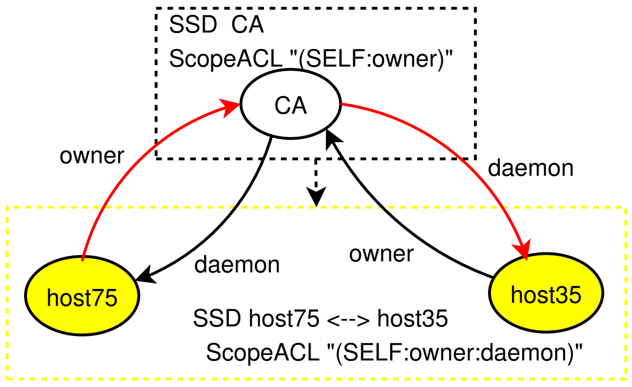


Figure 18: Security relationships of the default daemon credentials generated with ant.

Note that, as before, we can use the system property `SFHOSTNAME` to indicate a starting daemon in which sub-directory its credentials are. Moreover, we can change in the command line the sub-directory in which we create the credentials, e.g.:

```
ant  -DdaemonFromPrivate=host75 newDaemon
ant  -DdaemonFromPrivate=host35 newDaemon
```

What is the link between all these commands and the security model described in Section 3.1? Figure 18 shows what the security model looks like after we have created a "CA" and credentials for two daemons. By default daemons are assigned a binding with label `daemon` in the "CA's" local context. Also, a binding with label `owner` to the "CA's" public key is created in each of the daemons' local context. The result is that daemons can prove to each other the SPC `"(SELF:owner:daemon)"`, and therefore, they could share an SSD with that scope. Also, the "CA" can form its own SSD with the scope `"(SELF:owner)"`. Note that there is a clear ordering between both domains since the "CA" can also prove `"(SELF:owner:daemon)"`. Moreover, if all our ACLs only contain `"(SELF:owner)"` or `"(SELF:owner:daemon)"` SPCs, we can trivially verify all the desired SSD properties introduced in Section 3.4, i.e., **Consistency**, **Equivalence**, **Separability**, **Partial-Order**, **Granularity** and **Completeness**.

How can we mimic the security policy of the "old" implementation, i.e., a single trusted community? We could use the previous credentials and re-define the default `SSD` to be:

```
SSD extends LAZY NULL {
   ScopeACL "(SELF:owner:daemon)";
```

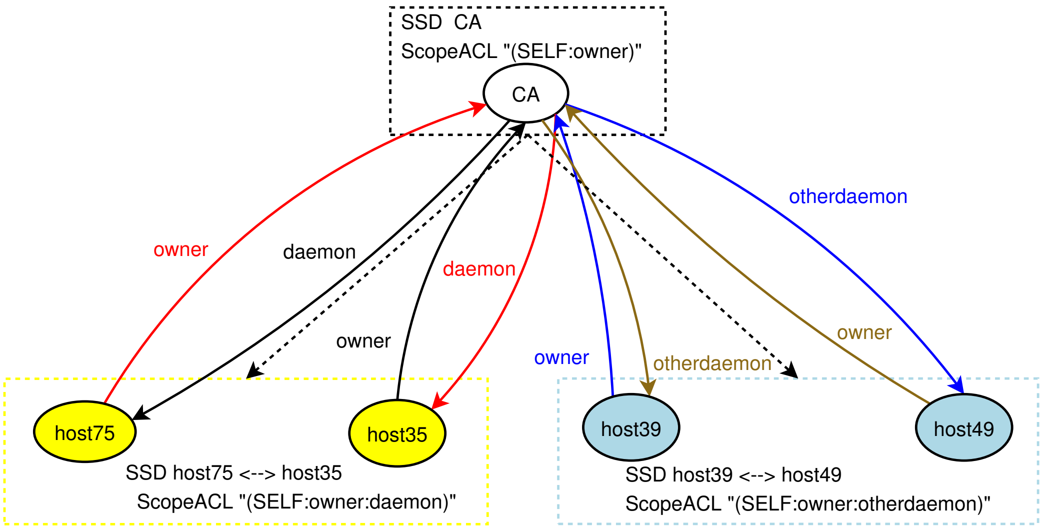


Figure 19: Security relationships of the default multi-domain configuration generated with ant.

```
        CallInACL ATTRIB ScopeACL;
        CallOutACL ATTRIB ScopeACL;
        CodeBaseACL ATTRIB ScopeACL;
        DeployDescriptionACL ATTRIB ScopeACL;
    }
```

How about creating multiple domains? We could change the label that the "CA" associates with the daemon's public key by specifying the java property "aliasDaemonKey", e.g.:

```
ant -DaliasDaemonKey=otherdaemon -DdaemonFromPrivate=host39 newDaemon
ant -DaliasDaemonKey=otherdaemon -DdaemonFromPrivate=host49 newDaemon
```

Figure 19 shows what the security relationships among principals look like after the previous commands. Now, we could create a new SSD defined by the `ScopeACL` "`(SELF:owner:otherdaemon)`" that is isolated from the previous domain, i.e., "`(SELF:owner:daemon)`", if we use the appropriate ACLs in the component descriptions. Note that we still satisfy all the desired properties of SSDs introduced in Section 3.4.

But you don't have to use Ant to create security credentials. Looking at the Ant configuration file in `private/buildSecurity.xml` you will see that the targets `newDaemon` or `initCA` are just calling the Java program `SFBuilderTool` with the appropriate arguments; so you could use these classes directly if you want to

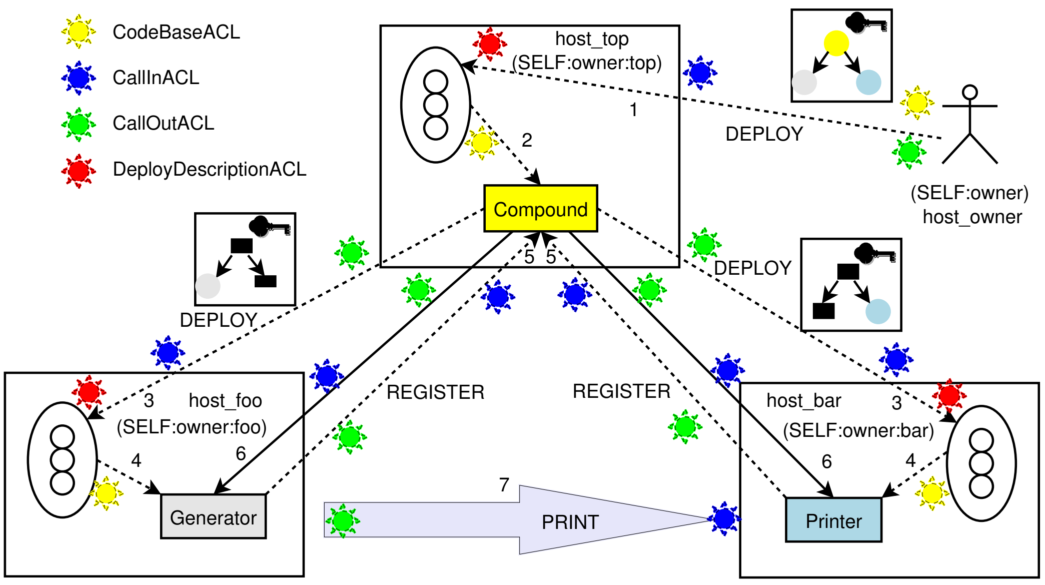


Figure 20: “Hello world” example

create your own tools to setup credentials. Also, there is nothing special about the “CA” credentials, so you could use any daemon’s credentials as the “CA”, e.g., by changing the property `CAFromPrivate` in the Ant command line, and increase the height of the delegation tree[14]. In fact, we could add a single edge with a label `myfriend`, from node `host39` to node `host49` by using the Ant command:

```
ant -DaliasDaemonKey=myfriend -DCAFromPrivate=host39
     -DdaemonFromPrivate=host49 newLink
```

However, it is easier to add edges dynamically by deploying special components that will delegate at run-time, as it is described in Section 6.2.

5.2 “Hello World” example

This first example is the only one explained in detail. However, in most cases this level of detail is not needed by the end user of the new security; therefore, this example will look a bit more complex than it really is...

[14]Note that the SmartFrog core classes are verified by using standard Java 2 security mechanisms and, according to the default java policy file, they should be signed by a key labelled `owner` (see Section 4.2). So you need to either re-sign the SmartFrog core jar file with the “new `owner`” key, or change the java policy file, or use a different label for the “parent” principal and add the “original `owner`“ key in the keystore...

Figure 20 shows a high level description of the "hello world" example, included with the new SmartFrog release, that is in the `org.smartfrog.examples.security.helloworld` package . We want to deploy three components: a "parent" component that implements the `Compound` interface, i.e., `compound`, and two "children" components that implement the `Prim` interface, i.e., `generator` and `printer`. We assume that they are deployed in three different nodes, i.e., `host_top`, `host_foo` and `host_bar`, respectively, and we want to enforce a different SSD in each of them. The behaviour of these components is as follows: after all the components get deployed, `generator` finds `printer` by using a lazy link that involves the "parent" `compound`; then, `generator` makes an RMI call to `printer` that "prints" something in node `host_bar`, e.g, "hello world".

The security complexity comes from trying to ensure that the "owner" that deploys the components is always in control of the nodes. For example, if any of the three nodes gets compromised, its credentials cannot be used to attack the other two by, e.g., deploying new descriptions, or changing the java classes that the other nodes execute. Moreover, we would like to enforce the "least-privilege" security principle, and, e.g., do not allow `printer` to invoke remote methods in `generator`. Let's describe first the steps needed to deploy these components securely:

1. The "owner" of the nodes uses the `sfStart` command to parse, sign, and start deploying a description in the remote node `host_top`; this deployment is started by propagating the signed description to the SmartFrog daemon running in that node.

2. The SmartFrog daemon in `host_top` starts deploying the description, creating the component `compound` locally.

3. The component `compound` triggers the remote deployment of its children, i.e., `generator` and `printer`, by forwarding the corresponding sub-descriptions to the SmartFrog daemons in nodes `host_foo` and `host_bar`.

4. The remote daemons in `host_foo` and `host_bar` deploy locally the sub-descriptions, creating the components `generator` and `printer`.

5. `Generator` and `printer` register themselves with their parent, i.e., `compound`.

6. `Compound` invokes its children's remote methods that are needed to complete their life-cycle.

7. `Generator` resolves the lazy link, finds `printer`, and starts invoking the "print" method on it.

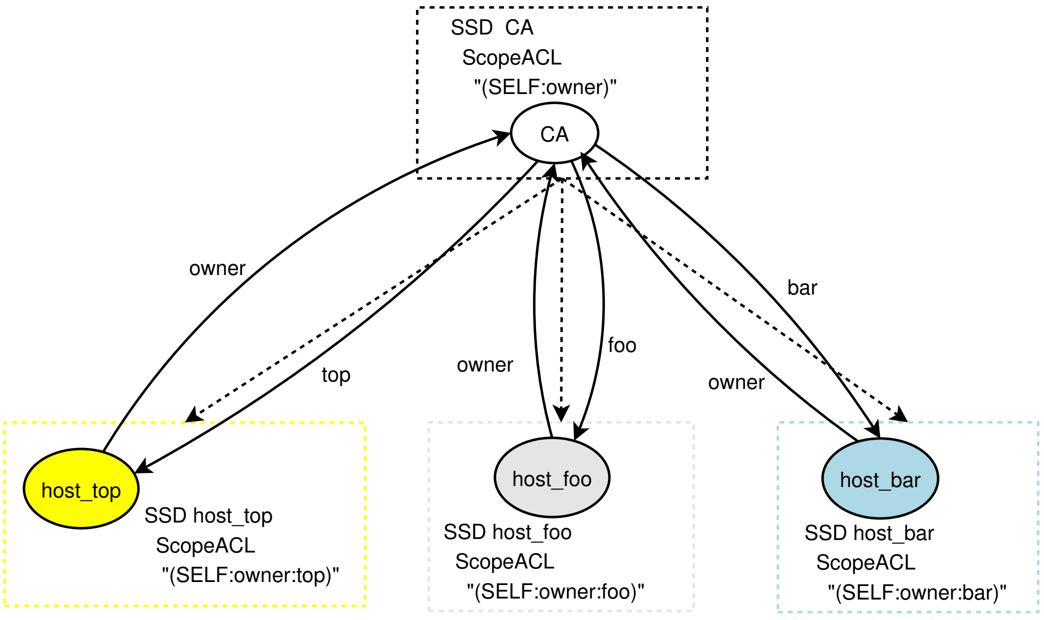


Figure 21: SSDs in the “hello world” example.

Now let’s look at what is the initial security set-up required to create independent SSDs in each host, i.e., as shown in Figure 21:

```
ant initCA
ant signJars
ant -DaliasDaemonKey=top -DdaemonFromPrivate=host_top newDaemon
ant -DaliasDaemonKey=foo -DdaemonFromPrivate=host_foo newDaemon
ant -DaliasDaemonKey=bar -DdaemonFromPrivate=host_bar newDaemon
%%and then for each daemon (assume shared file system...)
export SFHOSTNAME=host_top; ./bin/security/sfDaemon
export SFHOSTNAME=host_foo; ./bin/security/sfDaemon
export SFHOSTNAME=host_bar; ./bin/security/sfDaemon
%% and to start the deployment...
export SFHOSTNAME=CA; ./bin/security/sfStart <whatever...>
```

Let’s look in more detail at what interactions we need to allow between the SSDs in order to achieve the deployment in Figure 20:

1. The “owner” principal loads descriptions to be parsed from a signed jar file. Depending on which class loader loads this description, i.e., a `Codebase` or a `Classpath` class loaders (see Section 4.2), the signature associated with this jar file should either satisfy its `CodeBaseACL` or the default java policy,

i.e., signed by key labelled "owner". Then, it starts the deployment with an RMI call targeted to a remote `ProcessCompound` in node `host_top`; but in this case, both its own `CallOutACL` and the target's `CallInACL` will be satisfied since we implicitly relax policies for `ProcessCompounds` (see Section 4.1).

2. The daemon in `host_top` checks that the signature of the description satisfies its own `DeployDescriptionACL`. Then, it triggers a local deployment of `compound` that could require safe dynamic loading of classes, i.e., contained in a signed jar file satisfying its `CodeBaseACL`.

3. `Compound` performs RMI calls to deploy sub-descriptions, therefore, we need to verify matching `CallOutACL` and `CallInACL` pairs to daemons running in nodes `host_foo` and `host_bar`; this check is again implicitly relaxed since the targets are remote `ProcessCompounds`.

4. Both `host_foo` and `host_bar` daemons repeat the deployment checks of 2, i.e., `DeployDescriptionACL` and `CodeBaseACL` ACLs.

5. The registration of `generator` and `printer` with `compound` requires a `CallInACL` valid check into `compound,` and `CallOutACL` valid checks in the other two, i.e., `generator` and `printer`.

6. Life-cycle remote method invocations in `generator` and `printer` from `compound` require the reverse checks that we did in step 5.

7. `Generator` needs to verify a `CallOutACL` and the Printer a `CallInACL,` so that we can "print".

To summarise, every time we do an RMI call the source enforces its local `CallOutACL`, and the target validates a `CallInACL`; however, when the target is a `ProcessCompound`, i.e., a SmartFrog daemon, we implicitly relax the ACLs (see Section 4.1). Also, every time we deploy a component locally we need to verify the signature of its description with respect to `DeployDescriptionACL` and, if we load classes dynamically to deploy it, they should be in a signed jar file that satisfies `CodeBaseACL`. Finally, in order to register and start components we need bi-directional RMI calls, i.e., both `CallInACL` and `CallOutACL`, between parent and child. Let's look at the security annotations in the SmartFrog descriptions that allow these relationships:

```
compoundSSD  extends LAZY SSD {
  ScopeACL "(SELF:owner:top)";
  CallInACL ScopeACL ++ "OR" ++ ATTRIB generatorSSD:ScopeACL
            ++ "OR" ++ ATTRIB printerSSD:ScopeACL;
```

```
    CallOutACL CallInACL;
    CodeBaseACL "(SELF:owner)";
    DeployDescriptionACL CodeBaseACL;
  }
  generatorSSD extends LAZY SSD {
    ScopeACL "(SELF:owner:foo)";
    CallInACL ScopeACL ++ "OR" ++ ATTRIB compoundSSD:ScopeACL;
    CallOutACL ScopeACL ++ "OR" ++ ATTRIB compoundSSD:ScopeACL
               ++ ATTRIB printerSSD:ScopeACL;
    CodeBaseACL "(SELF:owner)";
    DeployDescriptionACL CodeBaseACL;
  }
  printerSSD extends LAZY SSD {
    ScopeACL "(SELF:owner:bar)";
    CallInACL ScopeACL ++ "OR" ++ ATTRIB compoundSSD:ScopeACL
              ++ ATTRIB generatorSSD:ScopeACL;
    CallOutACL ScopeACL ++ "OR" ++ ATTRIB compoundSSD:ScopeACL;
    CodeBaseACL "(SELF:owner)";
    DeployDescriptionACL CodeBaseACL;
   }
```

And we can associate these security domains with the previous components as follows:

```
  compound extends Compound {
    sfSSD compoundSSD;
    ...
    generator extends Generator {
      sfSSD generatorSSD;
      ...
    }
    printer extends Printer {
      sfSSD printerSSD;
      ...
    }
  }
```

5.3 "Hello World" example using a "gateway" component

In the previous example, the security relationships between components in different domains are not ideal. In order to allow `generator` and `printer` to register

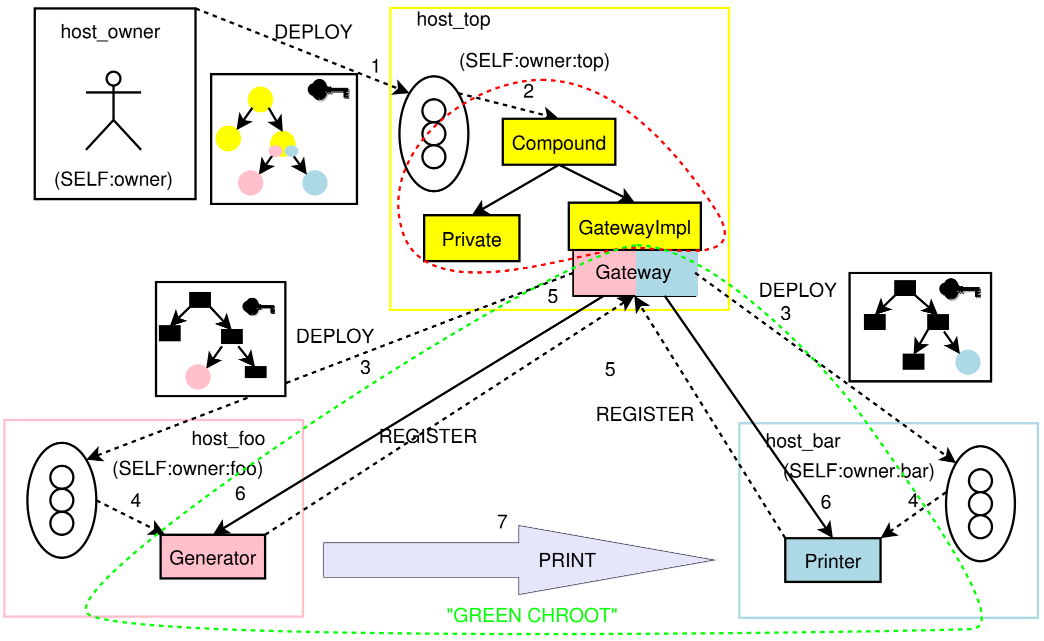


Figure 22: "Hello World" with gateway

with `compound`, all the components in the `compound's` domain need to allow incoming calls from both of them, i.e., add the SSD scopes of `generator` and `printer` to their `CallInACL`. Also, `compound` needs to propagate life-cycle actions to its children, therefore, all the components sharing domain with `compound` need to allow outgoing calls to both of them, i.e., add the SSD scopes of `generator` and `printer` to their `CallOutACL`. Clearly, enforcing a consistent policy for all the components in the same domain is exposing them to more risks than are needed. Hopefully, we could do better by using some of the techniques described in Section 3.5, that only allow access to methods in a remote interface of a particular component, i.e., a component implementing a "gateway" between SSDs.

Figure 22 shows an equivalent example to the one in Figure 20, but using the component `gateway`, implemented by the core SmartFrog class `GatewayImpl`, to communicate SSDs. This component implements a `Compound` interface, but also implements a `Gateway` interface, which contains just enough functionality for `generator` and `printer` to be safely parented by it, i.e., `Gateway` is a "low trust" interface and the use of `Compound` is restricted to the "high trust" domain. Moreover, the core class `GatewayImpl` makes more robust the default implementation of some methods in the `Gateway` interface, e.g.: remote references to `printer` and `generator` are explicit wrapped to allow outgoing RMI calls; changes to attributes in the `gateway's` context are tightly controlled; termination of a child does not immediately propagate up the component hierarchy; resolution of lazy links is "scoped", i.e., references outside the sub-tree rooted by gateway are not forwarded, creating the equivalent of a "chroot" file system.

However, we also want gateway to behave as a normal Compound when the request came from a component in its own SSD, e.g., the resolution of lazy links is no longer restricted, and this implies that most methods first check the origin of the request, and then change functionality accordingly (see Section 4.3 on how to do this check safely). For example, if generator requests gateway to resolve the lazy link "LAZY ROOT:ATTRIB printer", it will correctly return a remote reference to printer; however, if the generator′s request is the lazy link "LAZY PARENT:ATTRIB private" it will throw a resolution exception; but if private forwards this lazy link query to gateway, the answer will be a remote reference to private.

The result is that all the components that share SSD with gateway, no longer need to be accessible by printer and generator. For example, the component private, "sibling" of gateway, will not be accessible to them. Assuming a similar initial set-up to the previous example, the changes in the descriptions are as follows:

```
compoundSSD  extends LAZY SSD {
  ScopeACL "(SELF:owner:top)";
  CallInACL ScopeACL;
  CallOutACL CallInACL;
  CodeBaseACL "(SELF:owner)";
  DeployDescriptionACL CodeBaseACL;
}
// generatorSSD and printerSSD have not changed

gatewaySSD extends LAZY compoundSSD {
  ExplicitCallOutACL ATTRIB generatorSSD:ScopeACL ++ "OR"
                   ++ ATTRIB printerSSD:ScopeACL;
  ComponentInACL [
    ["org.smartfrog.sfcore.security.sf.Gateway",
     ATTRIB ExplicitCallOutACL]];
}
```

and now the annotation of component descriptions with SSDs looks like:

```
compound extends Compound {
  sfSSD compoundSSD;
  ...
  private extends Whatever {
      sfSSD compoundSSD;
      ...
    }
    gateway extends Gateway {
      sfSSD gatewaySSD;
      ...
      generator extends Generator {
        sfSSD generatorSSD;
        ...
      }
      printer extends Printer {
        sfSSD printerSSD;
        ...
      }
    }
  }
```

Although the security of SmartFrog components does not rely on a particular network topology, some applications that require "defence in depth" will benefit by combining SmartFrog "gateway" components with restricted network connectivity. For example, we could deploy `gateway` in a separate "bastion" host that has limited connectivity to the Internet, e.g., there is a TCP relay from a host in the DMZ to this host, and share LAN segment with the nodes where `private` and the root `compound` are deployed. Then, if we deploy `generator` and `printer` in other Internet domains, the previous example could work[15] without exposing `private` or the root `compound` to the Internet.

6 Advanced Topics

We anticipate that most users of the new SmartFrog security will just need the basic features described in Section 5, i.e.: they will statically create credentials for each principal using the "standard" Ant commands; they will use the SPCs that are naturally created by these commands to define SSDs, e.g., `"(SELF:owner:<whatever>)"`; they will use the default SPC, `"(SELF:owner)"`, to limit the signing of descriptions and jar files; they will use the core "gateway" components to restrict access from components in other SSDs. We hope that this should be reasonably easy to do, since it does not require much understanding of the underlying security model or changes to component code. However, when firewalls get in the way, certain

[15]We are assuming that `gateway` can open TCP connections across the Internet to `generator` and `printer`...

networking skills, not specific to SmartFrog, will be needed, e.g., configuring TCP relays or chosing wisely nodes and ports for “gateway” components.

Unfortunately, things get a bit more complicated when we try to fully exploit the new security features. In those cases, it is required a good understanding of the security model, and the implementation of some components may need security related code. It is our intention that, when we understand better how those advanced features are used, we will implement “higher level abstractions” for them, either by providing SmartFrog templates or new Java classes. However, since we will not get that critical feed-back until we have “advanced users”, we are trying to boot-strap the process by adding a few “teasers” to this document...

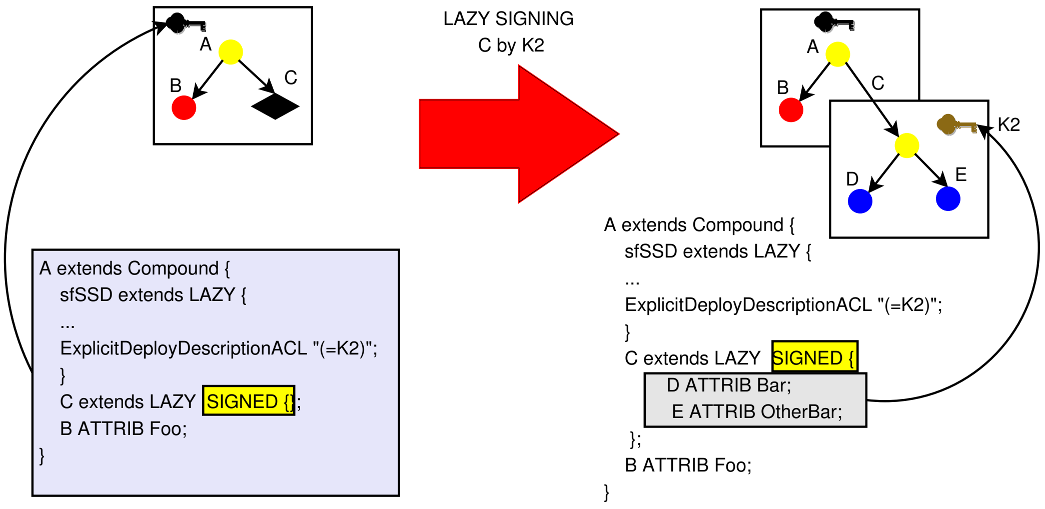
We will show next how to perform controlled delegation through either the lazy signing of sub-descriptions or the dynamic issuing of security credentials. In the first case, we use lazy signing to allow a “partially-trusted” scheduler to take allocation decisions on a set of tasks, but without changing the nature of those tasks. In the second case, we show how a common authority `A` can allow a third party `X` to take control of a node `Y`, so that `X` can replace `Y`’s security credentials by others that can validate a different SSD scope.

Finally, we show how to create a single federated SSD domain from a collection of resources that have credentials issued by different “local” authorities. This is not trivial when: we would like to add resources from a new authority dynamically and transparently, i.e., without changing the credentials of any resource currently in the domain; we would like to avoid issuing new credentials to every added resource, i.e., the credentials issued by “local” authorities are just fine; it should be easy to exclude current trusted “local” authorities from future SSDs, or to limit them to a subset of SSDs. The typical relevant scenario for this example is grid applications that obtain resources from different “providers” across the Internet.

You should view these examples as “patterns” that can be combined to implement more complex applications. For example, a “partially-trusted“ scheduler could allocate resources from different “providers” across the Internet, but we need to dynamically issue credentials for a “fresh” resource before it can be part of the “pool” of available resources...

6.1 Lazy signing of sub-descriptions

Lazy signing of sub-descriptions is supported in the SmartFrog language by qualifying with the keyword `SIGNED` a `"extends LAZY"` sub-description, i.e., a sub-description that contains “data”, and not new components that should be automatically deployed by the framework. During signing, when we reach a `SIGNED` sub-description, we stop the recursive hashing that we described in Section 4.1, so that anything changed inside that sub-description will not affect the signature of



```
A extends Compound {
    sfSSD extends LAZY {
    ...
    ExplicitDeployDescriptionACL "(=K2)";
    }
    C extends LAZY SIGNED {};
    B ATTRIB Foo;
}
```

```
A extends Compound {
    sfSSD extends LAZY {
    ...
    ExplicitDeployDescriptionACL "(=K2)";
    }
    C extends LAZY SIGNED {
        D ATTRIB Bar;
         E ATTRIB OtherBar;
     };
    B ATTRIB Foo;
}
```

Figure 23: Lazy signing of sub-descriptions

the enclosing description. Moreover, we expect that a `SIGNED` sub-description has attached to it a "self-contained" signature, i.e., a signature of just its own contents, and who can create that signature is specified in the `sfSSD` attribute of the immediately enclosing description, i.e., either by the inner attribute `DeployDescriptionACL` or the new `ExplicitDeployDescriptionACL`. Note that for "partial-trust" applications we recommend to use `ExplicitDeployDescriptionACL` and not `DeployDescriptionACL`, since allowing someone to modify a sub-description containing just "data", gives him less privileges than if he can chooses what components should be deployed, i.e., obtaining total control of the node.

Figure 23 shows an example of how to use lazy signing of sub-descriptions. The "owner" of the platform has created and signed a description that allows a principal with public key `K2` to further specify what the sub-description `C` contains, i.e., the SPC associated with the enclosing `ExplicitDeployDescriptionACL` is only valid for `K2`. This principal cannot modify anything else in the description, and, since `C` represents only "data", it cannot deploy arbitrary components in the nodes[16]. When we deploy that description, the target component, e.g., a `ProcessCompound`, should have an attribute `DeployDescriptionACL` that allows the "owner" to continue, but it does not need to know anything about `K2`, i.e., since signatures are checked recursively, any signed sub-description is validated with the enclosing ACLs, and the "owner" is effectively delegating authority to

[16]Some components use "data" sections to represent components that are explicitly deployed by them, e.g., some work-flow components. Declaring those sections "SIGNED" could allow arbitrary component deployments, if further checks are not done in the parent component before deploying its children...

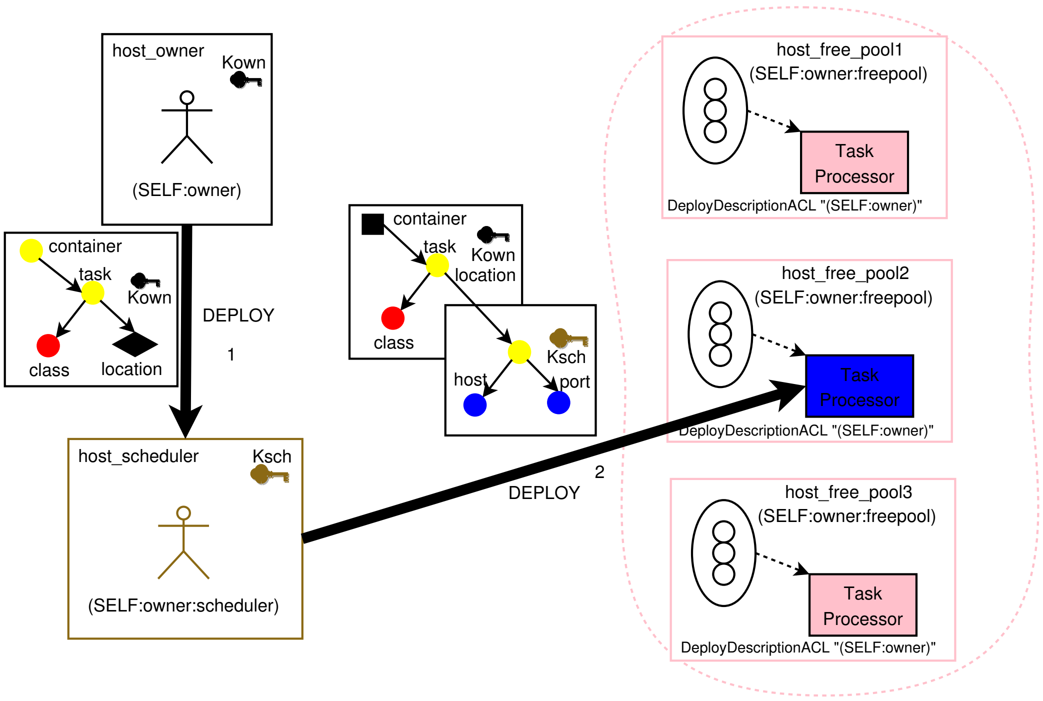


Figure 24: “Partially-trusted” scheduler

`K2` to sign `C`...

An important security consideration is that, since signatures of `SIGNED` sub-descriptions are “self-contained”, they should not rely on any external context in order to be “interpreted” appropriately. Otherwise, an attacker could take a signed sub-description and re-attach it in a different, but compatible, “entry point” or in a completely different description, and the “meaning” of what was signed will change. Therefore, we need to add attributes in the sub-description that capture this context, and, in most cases, application code should check that a sub-description is only used in the appropriate context. Unfortunately, the nature of this context is very application-dependent, and we find difficult to provide generic support in the framework for this task...

6.1.1 “Partially-trusted” scheduler example

Figure 24 shows an example of how to use lazy signing to minimise risk for critical sub-systems. Let’s assume that we have a set of tasks to do, represented by SmartFrog descriptions, and a pool of free nodes running SmartFrog daemons that could do them. There is an “owner” of the platform that has authority over all the nodes and he can validate the SPC `"(SELF:owner)"`. The job of allocating and deploying tasks to nodes in the pool is done by a component deployed in a separate node, that we call `scheduler`. We also assume that some customisation of the description is needed to allocate a task to a particular node, i.e., we need to add attributes to the sub-description `location`. We want to limit what descriptions can `scheduler` deploy so that, if its node gets compromised, all the nodes in the pool are not lost; in fact, we just want `scheduler` to modify location information and not the "nature" of the task to be deployed. We start by setting up credentials for `scheduler` and the nodes in the free pool, as we did in Section 5:

```
ant initCA
ant signJars
ant -DaliasDaemonKey=scheduler
      -DdaemonFromPrivate=host_scheduler newDaemon
ant -DaliasDaemonKey=freepool
      -DdaemonFromPrivate=host_free_pool1 newDaemon
ant -DaliasDaemonKey=freepool
      -DdaemonFromPrivate=host_free_pool2 newDaemon
```

and now the description signed by `"(SELF:owner)"` that grants `scheduler` the right to allocate tasks to "free pool" nodes will look like:

```
containerSSD extends LAZY SSD {
  ScopeACL "(SELF:owner:scheduler)";
  // we could use a gateway component instead ...
  CallInACL ScopeACL ++ "OR" ++ taskSSD:ScopeACL;
  CallOutACL CallInACL;
  // SSD defaults to "(SELF:owner)" for other ACLs
}
taskSSD extends LAZY SSD {
  ScopeACL "(SELF:owner:freepool)";
  CallInACL ScopeACL ++ "OR" ++ containerSSD:ScopeACL;
  CallOutACL CallInACL;
  ExplicitDeployDescriptionACL containerSSD:ScopeACL;
}
container extends TaskContainer {
  sfSSD containerSSD;
  task extends Task {
    sfSSD taskSSD;
    class extends LAZY TaskClass {
     // details that cannot be modified
    }
    // to be filled in by the scheduler
```

```
          location extends LAZY SIGNED {};
        }
      }
```

TaskContainer implements the Compound interface and uses a special "locator" component to find a target for its task; then, it generates and signs the location sub-description (as described in Section 4.1), replacing the location attribute in the description by this new sub-description; later, it continues with the deployment of its child, i.e., the task to be allocated to one "free" node. Note that TaskContainer could also implement a Gateway interface, and that could limit the exposure of scheduler to a malicious node in the free pool, as we saw in Section 5.3. Let's look at the final description forwarded by scheduler, and checked by the allocated node:

```
// signed by "(SELF:owner)"
...
task extends Task {
    sfSSD taskSSD;
    class extends LAZY TaskClass {
     // details that cannot be modified
    }
    location extends LAZY SIGNED {
      // signed by "(SELF:owner:scheduler)"
      nodeInfo <host, port, whatever task context...>;
     };
  }
```

Note that we are not specifying what "task context" is added to location, but in reality we should make sure that location is linked uniquely to the task, e.g., using a "global task identifier", and the allocated node should check before deployment that the identifiers in the task description and in location match. Otherwise, an attacker could change where tasks are deployed very easily...

6.2 Dynamic generation of security credentials

We use java classes to perform all security operations, i.e., create local bindings, issue certificates, configure the default certificate chain, reset daemons, and so on... These classes can be directly used by SmartFrog components so that security operations are linked to the deployment of special components. For example, we could split the deployment of an application into two phases: in the first phase, we deploy a set of components that change the security configuration of the nodes by, e.g., creating new bindings and distributing certificates; in the second phase, the "actual" application components are deployed but now they can assume a proper security configuration, e.g., all SSDs are properly defined. This "phased" deployment is extremely useful in a "utility provider" scenario, in which the initial credentials of a node given by the "infrastructure provider" need to be "translated" into credentials used in the application context.

`UpdateCredentialsImpl` is a core SmartFrog class that implements components that perform security operations in the node in which they are deployed. In particular, it allows us to specify in a description new local bindings to be created, existing bindings to be removed, changes to the default certificate chain used by the node, or even a new `sfSSD` that the top-level `ProcessCompound` should enforce after reset. Similar to `<PUBKEY>` SPCs, public keys in the description are represented by a base-64 encoded string of a self-signed certificate. When the component is started, it will validate and perform all the changes, triggering a full reset of the target daemon, i.e., `ProcessCompound`. For example:

```
changeNode extends UpdateCredentials {
  //add local bindings foo and bar
  addTrusted extends LAZY {
    foo "dsfsdfsdfdfsdsdfdsf";
    bar "2gh32g4jg32434g324g2";
  }
  // remove another binding for foo
  removeTrusted extends LAZY {
    foo "asdhahsdiuaydiy";
  }
  // set the default certs chain to this
  changeCertChain ["khkhkhkhkh","khkhkh","dhfhsdfhsd"];
  // change the deploy ACL in the ProcessCompound
  //   (leave the rest of the SSD the same)
  ssdChanges extends LAZY {
    DeployDescriptionACL "(SELF:owner:whatever)";
  }
}
```

When descriptions like the one above are not generated programmatically, it becomes cumbersome to type all these long base-64 encoded strings. We have added a new function (`patchkey`) in the SmartFrog language that "patches" the string-formatted public key[17] at parse time, if there is already a local binding with a given label in the security credentials used during parsing and signing, e.g.:

[17]We always use self-signed certificates to represent public keys, so we are really "patching" an encoded certificate...

```
changeNode extends UpdateCredentials {
   //add local bindings foo and bar
  addTrusted extends LAZY {
     // Assuming corresponding binding in my keystore
     //    gets replaced by foo "dsfsdfsdfdfsdsdfdsf";
    foo extends patchkey {
        alias "my_binding_for_foo";
        // do not format as an SPC...
        isRawFormat true;
    }
    bar extends patchkey {
        alias "my_binding_for_bar";
        isRawFormat true;
    }
    ...
```

We also use the same function to generate SPCs with directly embedded public keys, i.e., with `<PUBKEY>` elements. In this case, `isRawFormat` should be false, the attribute `rest` is concatenated to finish the SPC, and if there are multiple bindings for the same key label, we generate an `OR` of all the SPCs:

```
sfSSD extends LAZY SSD {
  // generates:
  // CallInACL "(asssassa:owner:hello)OR(sdads:owner:hello)"
  //  if I have two local bindings "foo", i.e.,
  //  with public keys asssasssa and sdads
  CallInACL extends patchkey {
       alias foo;
       isRawFormat false;
       rest "owner:hello";
  }
 ...
```

6.2.1 Node “take over” example.

Components that change credentials can be very useful, Figure 25 gives us a good example of it. Let’s assume that `free_pool` is a node[18] in the “free” pool, i.e., it has not been allocated to any task yet, and belongs to a SSD with scope

[18]By “node” we mean a physical node that is running a SmartFrog daemon, and it is identified by the credentials of that daemon...

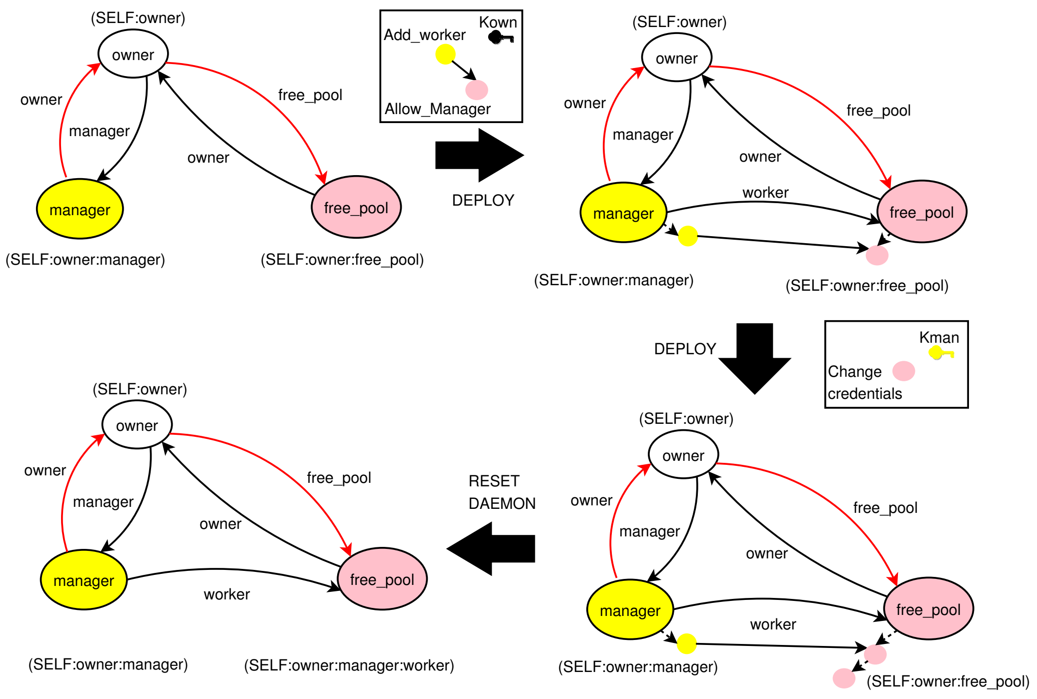


Figure 25: Node "take over" example

"(SELF:owner:free_pool)". Its "owner" is the node `owner`, i.e., a node belonging to SSD "(SELF:owner)", and it decides to "give it away" to a node `manager`, i.e., belonging to SSD with scope "(SELF:owner:manager)"; where by "giving it away" we mean that there is a full transfer of "ownership" to a different node. However, we would like to avoid `owner` having to directly interact with `free_pool` to make this transfer of ownership possible. Also, this operation could be required many times, e.g., if we are flexing resources according to load, and we would like to grant permission to `manager` to take control of any node in the SSD with scope "(SELF:owner:free_pool)", without further interacting with `owner`, e.g, to allow flexing of nodes with `owner` "off-line".

The process to do that, described in Figure 25, is as follows:

1. We start with a situation in which `owner` can sign descriptions that will be deployed in both `manager` and `free_pool`, but `manager` is not trusted at all by `free_pool`.

2. `Owner` signs and submits a description that deploys a "parent" component in `manager` and a "child" component in `free_pool`. The "parent" component finds out safely [19] the public key of `free_node` and creates an ap-

[19]We match the public key suggested by `free_pool` in the RMI call with the one used in the

propriate binding for it, i.e., with label "worker", in manager's local context. The "child" component is a "Trojan" component that implements the Compound interface, and it will allow manager to deploy "anything", as long as it is parented by this component, i.e., this component has the attribute DeployDescriptionACL set to "(SELF:owner:manager)".

3. The "parent" component deployed in manager constructs and signs, i.e., using manager's private key, a description with an UpdateCredentialsImpl component. This component description will contain: a new certificate chain that appends to manager's certificate chain the certificate associated with the newly created binding, i.e., allowing free_pool to prove the scope "(SELF:owner:manager:worker)"; a new binding with manager's public key that free_pool should add; and a change to the ProcessCompound SSD in free_pool that allows manager arbitrary component deployments.

4. Using as "parent" the "Trojan" component that implements Compound, previously deployed in free_node, it deploys the previous description and starts the UpdateCredentialsImpl component.

5. The starting component triggers a full reset of free_node that terminates all the local components and propagates termination to the parent component deployed in manager[20]. After free_node comes back from the reset, it will have the correct settings to be part of the SSD with scope "(SELF:owner:manager:worker)", and manager will have full control over it.

6. The original description signed by owner could be stored by manager and re-deployed to take over any other node of the "free" pool with similar scope[21], i.e., "(SELF:owner:free_pool)"[22]. Note that owner could limit this behaviour by having multiple pools of "free" nodes with different scopes, e.g., "(SELF:owner:free_pool_1_3)"...

We refer to the examples in the new SmartFrog distribution for details on the descriptions and components involved.

underlying SSL session.

[20]The "parent" component deployed in manager implements the Gateway interface, so this call back should be safe, and it is implicitly allowed.

[21]If the scope is different, the ScopeACL check for the "Trojan" component will fail, stopping deployment...

[22]We are assuming that the description did not contain location information, i.e., host name of the target, or this can be changed by manager without affecting signatures, as in the "partially-trusted" Scheduler example.

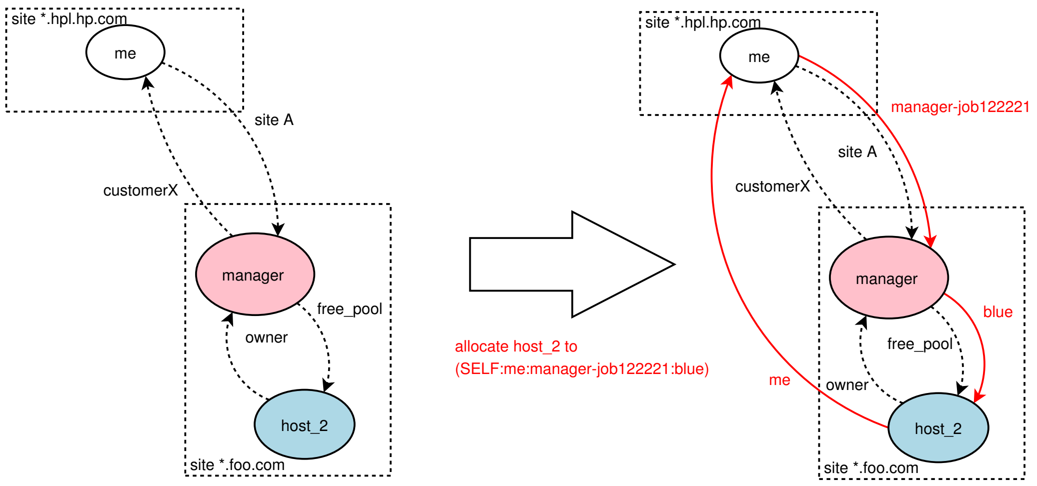


Figure 26: Allocating a new node from a federated site

6.3 Dynamic federation of a single SSD

In some cases we want to create a single SSD that have nodes provided by different local authorities. For example, computational grids that span across the Internet can have nodes hosted by different sites, but we want to make sure that they can cooperate among themselves regardless of where they are hosted. Similarly, we would like to deploy SmartFrog descriptions that create components in nodes across multiple sites, and they can prove to each other a similar SSD scope. Moreover, we would like to leverage credentials given by the "local" authorities so that we do not need to interact directly to set-up each node, i.e., we can be "off-line" during "flexing". Also, we would like to dynamically add new sites that are hosting our components, and they should be accepted in the SSD transparently, i.e., without changing the configuration of any existing members. Finally, we would like to have an easy way to exclude nodes of a site from certain SSDs or for any future deployments.

Figure 26 and Figure 27 show a possible solution to this problem. The principal `me` is the person that is deploying SmartFrog components across sites, i.e., it uses resources in different sites to host applications. Let's look at how `me` negotiates with different sites and obtains access to nodes:

- Each site has a principal `manager` that administers a pool of free local nodes, and is the main entry point for customers requesting nodes. Figure 26 shows how `me` and `manager` are mutually linked with arbitrary labels `siteA` and `customerX` to bootstrap their relationship. These links are created in the examples using Ant, but in reality they could involve "offline" transactions, probably involving money, between `me` and `manager`. Also, we assume that

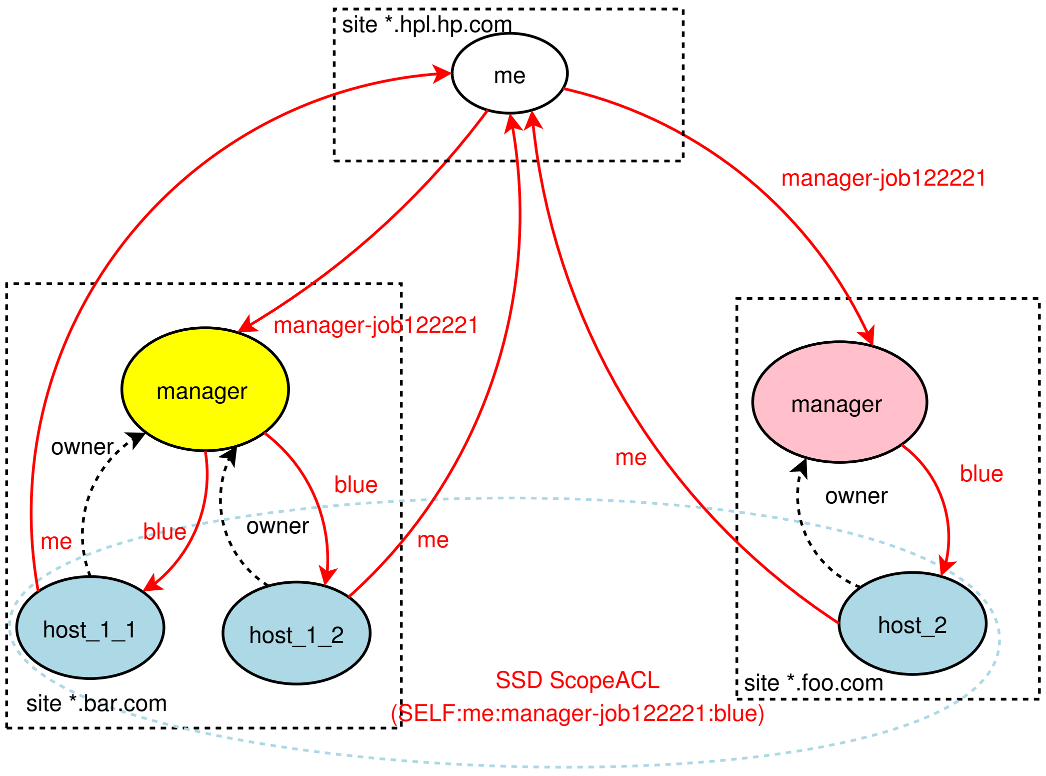


Figure 27: Dynamic federation of a single SSD

manager has full ownership on the nodes in the pool and identifies them with an arbitrary label `free_pool`.

- Figure 26 also shows how the graph changes when `me` requests a new node: it will instruct `manager` to use a particular label, e.g., "`blue`", that will define the scope of the SSD for that node. `Me` will also specify the name that the node should use to refer to it, e.g., "`me`" or "`joe.smith@foo.com`" or a base-64 hash of his public key or ....; this name should be consistent across sites, and a well-behaved site should not use the same label for a different customer. Also, in order to be consistent across sites, the new node will create a link to `manager` with label `owner`, if a different naming convention was initially used by `manager`. Finally, `me` will create, if needed, a local binding with a particular label, i.e., "`manager-job122221`", linked to `manager`. This label is a "session" label that will be shared by all the sites deploying components in this "session". We will discuss later how `me` chooses it.

- `Manager` will change the ownership of the allocated node using a mechanism similar to the one described in Section 6.2. In particular, `me` will be the new "owner", and a certificate chain that allows the node to prove the scope "`(SELF:me:manager-job122221:blue)`" will be added. Note that this chain does not require that `me` creates new bindings, i.e., only `manager` issues new certificates, and `me` does not need to be "on-line" to provision an extra node in that site...

- In Figure 27 `me` has repeated the previous process with a different site and now the nodes `host_1_1`, `host_1_2` and `host_2` form a single SSD with scope "`(SELF:me:  manager-job122221:blue)`". We can use this domain in a description as follows:

```
wideBlueSSD extends LAZY SSD {
  ScopeACL "(SELF:me:manager-job122221:blue)";
  /* We want "me" to take control,
       but without signing core classes...*/
  CodeBaseACL "(SELF:me) OR (SELF:owner)";
  DeployDescriptionACL CodeBaseACL;
}
```

and we can have SmartFrog descriptions that are deployed across sites transparently:

```
all extends Compound {
        //deploy in node at site bar.com
        sfSSD wideBlueSSD;
         printer extends Printer {
          //deploy in node at site foo.com
          sfSSD wideBlueSSD;
          ...
         }
        ...
     }
```

What if we need multiple SSDs?. We have several options: if we want to ensure that a less trusted site cannot host our "high trust" domain, we just change the "`manager-job122...`" label for our "high trust site" to, e.g., "`manager-high-job122..`", and define the new scope for that domain as "`(SELF:me:manager-high-job122221:blue)`". For example:

```
     wideHighTrustBlue extends SSD {
       ScopeACL "(SELF:me:manager-high-job122221:blue)";
       // we want "me" to take full control...
       CodeBaseACL "(SELF:me) OR (SELF:owner)";
       DeployDescriptionACL CodeBaseACL;
     }
```

Alternatively, if the "flexing API" that the site provides to allocate new nodes allows you to configure the label, we could ask for a "`red`" node, i.e., one that can prove the scope "`(SELF:me:manager-job122221:red)`" or we could dynamically change its credentials after ownership changes.

The main reason for using a "session" label to bind to a site is that we can isolate future deployments from that site, if we never reuse the same label and we do not create a new binding for it. This provides a "brute force" site revocation method that could involve re-deploying an existing application with a new label...

To summarise, we can create a single (or multiple) domains that span across dynamically added sites; we do not need to issue credentials directly to each node; we can revoke a whole site by changing a "session" label or restrict a domain to a subset of sites; and all of this with minimal impact to the SmartFrog component descriptions...

7 Conclusions

We have described the new features available in SmartFrog to secure interactions between components. We started with an abstract, graph-based representation of security relationships and use this model to constrain remote interactions of SmartFrog components. We use the SmartFrog language to annotate component descriptions with these constraints. Then, we looked at how these constraints are actually enforced in our framework, e.g., how component deployment uses lazy signing mechanisms, how to use secure dynamic class loading, how secure channels are established for RMI interactions... Finally, we use examples to give you some intuition on what can we do with this framework; first, a set of examples that do not require much understanding of the model, but will be useful to most users; then, more complex examples that show the real power of the framework...

Acknowledgements

I would like to thank the other members of the Serrano team, i.e., Peter Toft, Paul Murray, Julio Guijarro, Steve Loughran and Patrick Goldsack, for critical support in the development of this research. In particular, dynamic discussions with Patrick Goldsack heavily influenced most aspects of this work. I also would like to thank Brian Monahan from TSL for earlier discussions on formal modelling and tool support for SmartFrog security.

References

[1] Li Gong. *Inside Java 2 Platform Security*. The Java Series. Addison Wesley, 1999.

[2] Erik Hatcher and Steve Loughran. *Java development with Ant*. Manning, 2003.

[3] Hewlett-Packard. *The SmartFrog Reference Manual*, July 2005.

[4] Antonio Lain and Patrick Goldsack. Securing the life-cycle management of distributed components. Technical report, HP Labs, 2006. in preparation.

[5] Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, 1995.

[6] Ronald L. Rivest and Butler Lampson. SDSI – A simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession, April 1996. SDSI Version 1.0.