HiPoLDS: A Security Policy Language for Distributed Systems

Matteo Dell'amico, Gabriel Serme, Muhammad Idrees, Anderson Santana de Olivera, Yves Roudier

HiPoLDS: A Security Policy Language for Distributed Systems

Matteo Dell'Amico[1], Gabriel Serme[2], Muhammad Sabir Idrees[1], Anderson Santana de Olivera[2], and Yves Roudier[1]

[1] Eurecom, Sophia-Antipolis, France
[2] SAP Research, Sophia-Antipolis, France

Abstract. Expressing security policies to govern distributed systems is a complex and error-prone task. Policies are hard to understand, often expressed with unfriendly syntax, making it difficult to security administrators and to business analysts to create intelligible specifications. We introduce the Hierarchical Policy Language for Distributed Systems (HiPoLDS). HiPoLDS has been designed to enable the specification of security policies in distributed systems in a concise, readable, and extensible way. HiPoLDS's design focuses on decentralized execution environments under the control of multiple stakeholders. Policy enforcement employs distributed reference monitors who control the flow of information between services. HiPoLDS allows the definition of both *abstract* and *concrete* policies, expressing respectively high-level properties required and concrete implementation details to be ultimately introduced into the service implementation.

1 Introduction

Service-oriented architectures (SOAs) are a major software development pattern that builds applications based on loosely coupled services which can be run by different entities. Because of their complexity and of the varying degrees of trust between locations in which code is deployed and executed, it is challenging to make these systems secure. In particular, security is a *crosscutting* requirement: security-related code is generally scattered over several pieces of code and locations. What is worse, a local vulnerability or a mismatch between the security mechanisms adopted at different location can have dire consequences, potentially putting large systems at stake.

The CESSA project[3] focuses on the daunting task of making large SOAs secure by using aspect-oriented structuring and modularizing security across administrative and technological domains. It is with this goal in mind that we introduce the HiPoLDS security policy language. This language aims at being an efficient tool to express policies in diverse and complex distributed systems, where several entities interact in complex scenarios. SOAs are our motivating use case and they have driven our design, but HiPoLDS has been designed to be applicable to any kind of distributed system.

[3] `http://cessa.gforge.inria.fr`

HiPoLDS provides the following features which are not present together in existing security policy languages:

- Allowing to describe the security policy also by way of *abstract* requirements: this should allow the writer of the policy to mention *which* security properties they want (*e.g.*, confidentiality, authentication, *etc.*) along with how they are implemented (*e.g.*, encryption, signatures. *etc.*).
- Expressing the security policy of a service-oriented architecture centrally despite its decentralized enforcement. A single abstract requirement (*e.g.*, confidentiality or authentication) often needs to be implemented distributedly with several pieces of code at different locations (*e.g.* encrypt somewhere and decrypt in another place, or analogously for signature and verification). HiPoLDS aims to make the relationship between the abstract requirement and its distributed implementation clear.
- Keeping specifications clear and understandable, minimizing the need for code duplication and helping maintainability – even when policies are drafted cooperatively by several entities.

After discussing the state of the art on distributed policy languages in Section 2, we introduce the main constructs of HiPoLDS (Section 3), which is based on a hierarchy of *policy domains*, each of them being a set of locations at which security policies apply. HiPoLDS security rules are handled exclusively by reference monitors (RMs) running at each policy domain and controlling the flow of information crossing their borders. We show various use cases highlighting how HiPoLDS makes it easier to express complex security requirements, and relate security mechanisms with them (Section 4).

HiPoLDS has been designed with a top-down approach, by taking into account concrete practical use cases and deriving the features that were needed in such situations [9,10]. We discuss our plans towards a complete HiPoLDS implementation (Section 5), also discussing automated and semi-automated strategies to relate requirements with mechanisms that implement them. We conclude (Section 6) by highlighting open research issues related to HiPoLDS and discussing our agenda for future research.

2 Related Work

The expression of a security policy is central when it comes to describing how to secure a system. We review in this section a few of these approaches and in particular how appropriate they are for the distributed deployment of a service-oriented architecture. A large number of security policy specifications aim at mediating and restricting access to a central database. Those approaches cannot qualify for SOAs due to their distributed nature. In addition, the security policy of a SOA has to capture responsibilities about the enforcement of the security policy and the fact that not all execution environments where services are running can be controlled.

Even in decentralized settings, access control policies have generally been well covered. The SecPal language [2] is one such proposal for describing decentralized access control which formalizes the use of SPKI certificates. It is interesting in that, similarly to the underlying PKI infrastructure, it captures rather well the notion of trusted authorities and their respective competences for authentication. However, SecPal expressions are restricted to the access control model to be enforced and cannot describe any manipulation of messages required for more complex security policies. This means it also would not be very appropriate for analyzing complex and extensible protocols like those encountered in service-oriented architectures. Contrary to SecPal, some policy languages aim at extensibility rather than formal verifiability; this is the case of Li *et al.*'s approach [14]. The policy model is captured through facts and inference rules, which may be interesting for introducing additional concerns.

Information centric approaches like the Decentralized Label Model [16] or its variants aim at specifying formally the security properties of the information flow in a system. This description is implemented through the typing of information flows with labels, in particular confidentiality labels in the case of Myers and Liskov's label model. One advantage of this approach is that it is also very declarative and may describe many different properties beyond access control. Implementing policy enforcement may however be rather difficult to implement: some automation is required when moving to low-level operations, in particular with respect to the selection of the cryptographic mechanisms used and to the key distribution operations. All of those are left outside the security policy specification, thus likely preventing the customized combination of multiple encryption techniques; it is also implicitly assumed that this implementation will be "correctly" deployed, whatever that may mean to the security expert. Furthermore, while they are simple because of their high-level of abstraction, information flow security models require handling the declassification of information, which more or less breaks the regularity of the policy. Still, the DLM is at an advantage here compared with similar models by making this declassification operation explicitly described in the language.

It is worth noting that the security policy specifications described above approach the expression of the policy as a high level statement of security objectives or properties for the sake of separating the policy model from its implementation. However, by not considering low-level concerns related to policy enforcement they also fail to capture network boundaries, network domains, and the protocols between them, all which are however extremely important for the specification of relevant policies.

In contrast, the SPL language [21,20] is quite inspirational in that it expresses the distributed enforcement of obligation policies at different levels of abstraction. Those policies easily map to reference monitors for enforcement. SPL also aims at providing a unifying framework for policies expressed at multiple places in a company. Still, SPL assumes that the enforcement is performed by a trusted entity which is not adapted for addressing SOA security in general. Furthermore, the policies expressed in SPL are simply access control related in that some in-

formation is authorized or forbidden, the expression of that requirement being essentially focused on the description of the reference monitor operation.

Ponder [7] is another language based on obligation and filtering whose expressivity is more extended. It too fails to express the existence of multiple entities for enforcement.

The Law Governed Interactions approach [23] (LGI) also constitutes a very interesting attempt at specifying policies over multiple domains, like network domains. LGI aims at rather diverse types of policies, even beyond security ones, encompassing for instance quality of service concerns. Policy enforcement in LGI is based on the realization of a policy based middleware in which communication is mediated by reference monitors between domains. In this approach, domains can be considered as governed by a mandatory policy, their law. However, the approach fails short to account for multiple stakeholders by not considering that the enforcement might not always be possible - or at least not by an authority that is trusted enough to ensure the application of the law. Unlike LGI, in HiPoLDS reference monitor do not need to be trusted by all participating entities, and need to be as trusted as the applications running in their domains.

All three approaches above feature the idea that the concept of domains is not only central to enforcement by an associated reference monitor but also central to the very specification of security policies. Our work builds on the idea that a domain does not only mean a consistent policy is enforced, but that the enforcement is under the control of a single authority. Given that authority model, this architecture may to some extent also help solve policy composition issues [1], even though we do not address this issue in this paper.

Using AOP for policy enforcement is not an original idea in itself. Several works apply AOP for building inline reference monitors for access control, such as [19,25,8] or to specify other requirements such as availability [6]. The particularity in our approach is its application to the information flow in a distributed systems, which requires the use of specific aspect mechanisms [13].

Hierarchical policies also require another approach to policy composition and conflict resolution. In previous work, conflicts are mostly solved by disambiguating among diverging policy decisions [3,15,11]. We advocate that the hierarchical organization of HiPoLDS policies essentially impacts the enabled information flows between domains controlled by different authorities. Handling such issues rather requires advanced negotiation techniques when policies cross domains.

3 Language Overview

In service oriented architectures, complex processes are carried out by several interacting entities. It is essential to support a concise way of specifying high-level policies that need to be applied in the whole architecture or in large parts of it, as well as fine-grained requirements that need to be applied in smaller domains. In HiPoLDS, we do this by defining the whole system architecture as a hierarchical structure of *policy domains*, and using *reference monitors* that enforce security policies at the border of policy domains.

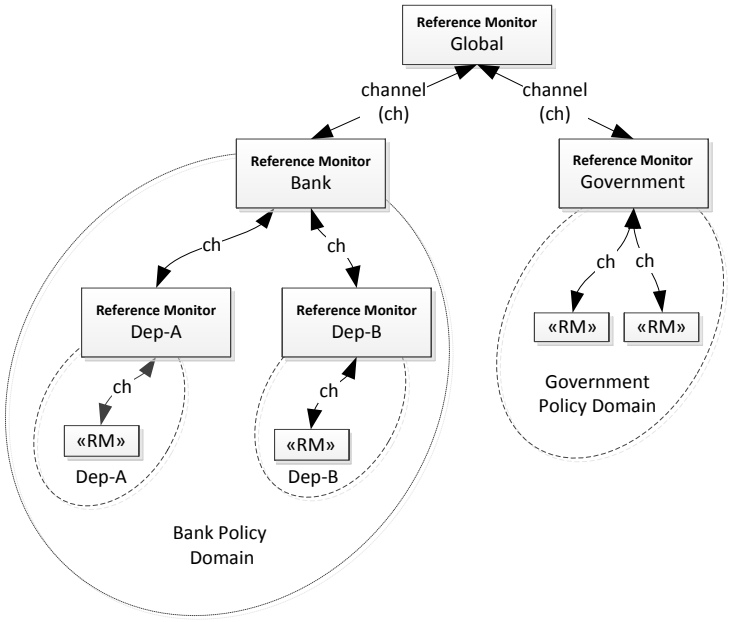

Fig. 1: Domain Hierarchy Example

A HiPoLDS document is composed of two parts: a declaration of the domain hierarchy, and a set of rules. The domain hierarchy, complemented by domain attributes, describes the global system architecture, while rules describe security policies that will take place. In this Section, we will introduce them along with the syntax, by means of simple examples.

3.1 Policy Domains

Policy domains can represent very different entities, such as corporations, individuals, down to the level of real or virtual machines, or even applications running on those machines. They are the scope for HiPoLDS rules, and they correspond to sets of entities that have particular properties that can be relevant to security (for example, a security domain can correspond to an organization, an individual, a place, a real machine, a virtual machine, and even a single application). Leaves of a policy domain hierarchy should be small enough that no security enforcement is needed for communications within a single policy domain. This is illustrated in Figure 1, which shows an example hierarchy for a loan negotiation scenario, where a bank communicates with a governmental information system in order to evaluate the eligibility of its customers to loan aids. We will use this scenario as a running example throughout this paper.[4]

[4] A complete HiPoLDS specification for this scenario is available in [9].

Policies regulate the exchange of information between different domains. They are enforced and monitored *at the borders* of policy domains. In this way it is possible to make sure that, for example, particular information sets remain confined within a domain (*e.g.*, to fulfill a privacy requirement) or get annotated with additional security metadata (*e.g.*, a Message Authentication Code or a cryptographic signature). The hierarchical structuring of policy domains allows the coexistence of policies that cover a wide set of locations (such as an organization) and of very specific ones (for example, access to a particular service).

Any domain can contain any number of subdomains, up to an arbitrary level of nesting. For a domain included within a parent domain, policies both in the enclosing and the inner domains will apply. This allows us to naturally define rules both at large (*e.g.*, organizations) and small (*e.g.*, services) scales. The hierarchy of policy domains also allows drafting a security policy cooperatively: the language supports the definition of rules that only apply within one, or more, policy domains. In our framework, administrators from involved entities should agree on "top-level" security policies applying to all domains, but they can be free to define additional security policies applying only to a domain of their competence. Data fields can be attached to policy domains, for example to contain encryption keys used by a principal.

In very complex scenarios, the number of policy domains might become very large, making the policy domain hierarchy unwieldy to handle manually. Currently, HiPoLDS does not handle this problem, which can be however managed by resorting to an external macro language such as, for example, GNU M4.[5]

In a deployed implementation, the definition of the domain hierarchy should also include a way to match the addressing scheme used for communication between services with the policy domains, in order to allow reference monitors to evaluate the origin and destination policy domain for a message.

Domain Attributes We allow specifying an arbitrary number of additional *attributes* for each policy domain, when defining the policy domain hierarchy. Domain attributes are text labels allowing to attach additional information to the containment relationships implied by the hierarchical domain policy structure.

Domain attributes allow to specify policies that apply to several policy domains without the need to list them explicitly, allowing them to "cross" the policy domain hierarchy. For example, all policy domains corresponding to mobile devices can be labeled as "mobile"; afterwards, it will be easy to define policies that apply to all mobile devices in the policy domain hierarchy, or some sub-hierarchy of it, by writing policies that apply only to domains with a "mobile" attribute. Such a feature is essential for avoiding repetitions and keeping rules as terse as possible.

Domain attributes can be used for several purposes, such as describing what a policy domain corresponds to (*e.g.*, an organization, an individual or a device) or some technical architecture details (*e.g.*, the kind of operating system running on a machine).

[5] `http://www.gnu.org/s/m4/`

All such information are potentially relevant with respect to the security policies, and domain attributes can be used to specify policies that apply to domains with common characteristics, even in different parts of the domain hierarchy. Using policy domain attributes increases maintainability: when a new policy domain is created, it will be sufficient to label it with the appropriate attributes and the relevant security policies will be applied to it as well.

Example We show the HiPoLDS declaration of part of the domain hierarchy for the loan negotiation scenario. The nesting of policy domains is expressed by enclosing inner domains in curly braces, and for each domain the list of domain attributes is written as comma-separated between parentheses. Here, domain attributes make it possible to differentiate the clerks from the manager. For brevity, we omit data items attached to domains, such as for example the encryption key IDs used by the bank employees.

```
Bank (organization) {
    Dept-A (department, organization)
    {
        employee (manager),
        Sub-department (subdomain, department),
        {...},
    }

    Dept-B (department, organization)
    {
        employee (clerk),
        ...
    }
}
```

3.2 Reference Monitors

In our model, security is at stake if proper measures are not taken when information traverses policy domains. For example, due to a confidentiality requirement, one or more pieces of information should not be readable outside a given set of policy domains, and this requirement is not fulfilled when the information is sent unencrypted outside of the allowed domains, or the encryption keys are divulged. In our system model, a *reference monitor* per domain monitors all the information entering or exiting the domain and alters it as needed. A reference monitor communicates exclusively with the services in its own domain and with the reference monitors of the neighbors in the domain hierarchy (*i.e.*, parent and child domains).

Reference monitors are as trusted as anything in their policy domain: rather than being a trusted infrastructure (as, for example, in LGI), they are simply used to enforce and/or monitor the security mechanisms, separating them when possible from the business logic of the application. Reference monitors intercept, and take action, on communications across trust domain boundaries. They work

similarly to "customs control", enforcing restrictions about what gets in and out of a domain. Some actions that reference monitors can apply are:

- filtering: reference monitors can implement access control to resources outside of their original policy domain[6] by filtering unauthorized messages; +
- cryptography: information can be encrypted, decrypted or signed when leaving or entering policy domains;
- managing security metadata: in our system, information is augmented with metadata that we label as *information tags* (see Section 3.4).

In other cases, reference monitors can enforce security policies by triggering actions that will take place in their policy domain.

3.3 HiPoLDS Rules

Rules are the way in which security requirements are specified in HiPoLDS. The form of a rule is `SCOPE {LEFTPART → RIGHTPART}`.

- The **scope** identifies the part(s) of the policy domain hierarchy in which the rule needs to be enforced. If omitted, the default scope for a rule is the whole domain hierarchy.
- The **left part** is a set of comma-separated clauses that describe the conditions that trigger rule enforcement. The rule is enforced when all the clauses on the left part are true.
- The **right part** describes the properties that are required to hold. The rule is satisfied when all the clauses on the right part are true.

A first example of a rule is the following one:

$$\texttt{x} \rightarrow \texttt{x}\ \mathit{is}\ \texttt{confidential}(\mathit{Bank}, \mathit{Government})$$

The scope here is omitted, meaning that the rule applies to the whole policy domain hierarchy. The left part of the rule, in this case, matches the only variable x. In HiPoLDS, variables match pieces of information; if they appear on the left side, they are implicitly quantified universally. In this case, the variable x therefore matches any piece of information exchanged in the whole domain hierarchy. If a variable appears only on the right side of the rule, it is instead implicitly quantified existentially, meaning that security enforcement mechanism must ensure that such an assignment to the variable exists such that the right part of the rule is satisfied.

On the right side, *confidential* is a security property – specified by a list of domains – that must be ensured. Security properties in HiPoLDS are preceded by

[6] Policy domains can be made as small as required; for example, to enforce access control to a service from any other location, a policy domain can enclose only the original service.

the **is** keyword. The *confidential* property requires that pieces of information will not be readable outside of the specified policy domains – Bank and Government, in this case.

This particular rule requires that the *confidential* property is ensured, but it is underspecified, in the sense that it can be implemented in different ways. For example, all messages that leave Bank or Government can be encrypted with keys only available in those domains, or messages can just be filtered when they leave any of those domains. We refer to rules that can be implemented in several possible ways as *abstract* rules, as opposed to *concrete* rules that give a complete specification that can be executed by reference monitors. Since abstract rules are less verbose and more focused on the security properties that are needed, we consider the ability to express them as very beneficial towards having clear and maintainable security policies. The development of inference techniques that would help writers of security policies derive concrete rules starting from high-level, abstract ones is currently an open issue on which we are still working.

Rules are *not monotonic*. For example, consider a sub-domain B of domain A. If we consider requirements on data confidentiality, a piece of information can be allowed to be readable only within B and not in the rest of A, but also the opposite can apply: if B for some reason is considered "less trusted" than the rest of A (*e.g.*, a mobile device that can fall more easily in the hands of an attacker), then restrictive rules can be applied whenever some data is sent to B.

It is possible that rules will require actions that are impossible to satisfy or in conflict with other rules. We plan to investigate how to detect conflicting rules, both statically (*i.e.*, when drafting the security policy) and at runtime, and on determining ways to manage them. Conflicts within a policy domain are the easiest to solve as they only correspond to local policies as defined by the same authority. In the current proposal, we limit ourselves to an order-based prioritization of rules within a policy domain, in the style of most firewall policies; other approaches for solving conflicts have been vastly explored in the literature and might be applied as well. In contrast, conflicts between policies defined in different domains are harder to solve as they are defined by potentially different authorities and thus require some negotiation. Due to the style of HiPoLDS policies which only adds further security constraints to the diffusion of information flows between policy domains, those conflicts cannot increase the rights granted to principals; instead they may impede communication between two policy domains, especially if an intermediate domain prohibits information to flow across. In this current proposal, we will limit ourselves to a simple priority rule, by choosing the more specific rules (*i.e.*, those defined for inner policy domains) and, to discriminate between rules defined at the same hierarchy level, we will give priority to the one defined first.

3.4 Information Tags

Information tags are free-form text labels representing some security metadata that is attached to information and categorizes it. It is possible to define

HiPoLDS rules that apply to information that has particular tags; when combined with domain attributes this allows us to naturally define policies that apply to large sets of information and span different policy domains. Reference monitors manage information tags and use information tags to decide which actions to take. For example, based on the tags it has, information can be filtered, transformed (*e.g.*, through encryption, stripping of confidential information, sanitization against injection attacks) and/or rerouted. Information tags are stripped before sending information to the original services, which will behave as if no security mechanisms were put in place. The following example shows how an information tag can be used in HiPoLDS.

$$\begin{array}{l} \texttt{m} : \mathit{message},\ \texttt{x} : \mathit{customer\text{-}info} \in \texttt{m}.\mathit{contents}, \\ \texttt{m}.\mathit{from} == \texttt{y} :: \mathit{employee},\ \texttt{y}.\mathit{key} == P_k \qquad \rightarrow \texttt{x}\ \mathit{is}\ \texttt{signed}(P_k) \end{array} \tag{1}$$

In this case, the scope of the rule is the `Bank` domain. In its place, a domain attribute could have been there (for example, `bank`) to specify that the rule applies to all banks in the domain hierarchy. In this rule, we see for the first time the '`:`' and '`::`' constructs, which are used respectively to match variables representing data with information tags and those representing domains with domain attributes. In addition, data fields on information and on policy domains are accessed via the dot notation seen in `m.contents` and `m.from`.

In this case, the left side of the rule uses information tags and domain attributes to match any message m sent from an employee y in the Bank. Since variables appearing on the left side of the rule are quantified universally, the contents of the message m are bound to x. Then y's key is bound to the variable P_k; finally, the right side of the rule requires that the message is signed with the key P_k. When the message is sent, reference monitors will verify the state of the message; if the left side matches and the right side does not (*i.e.*, the message is not signed) the appropriate reference monitor can add the signature – if it has access to the private key P_k – or drop the message. In summary, the rule above can be read as follows in plain English: *"The following rule applies only to the policy domain of bank. For each message m sent by an employee y with a public key P_k, P_k must be used to sign the contents of the message."*

4 Examples

After introducing the basic structure of HiPoLDS, we now show how its features can be adopted in a realistic case. We illustrate the relationship between concrete and abstract rules, discuss the role of reference monitors, and show how domain attributes can be used to describe role-based security rules. We will continue our discussion using the loan negotiation example from the previous section.

4.1 Abstract and Concrete Rules

Let us consider a security requirement of the following form: *"Customers' private information should only be disclosed to the Bank and the Government, and*

its integrity has to be guaranteed". Such a requirement would translate to the following HiPoLDS *abstract* rule:

$$\begin{array}{l} \texttt{m} : \mathit{message},\ \texttt{x} : \mathit{customer\text{-}info}\ \epsilon\ \texttt{m}.\mathit{contents} \\ \rightarrow \texttt{x}\ \mathit{is}\ \texttt{confidential}(\mathit{Bank}, \mathit{Government}),\ \texttt{m}\ \mathit{is}\ \texttt{integrity\_verified} \end{array} \tag{2}$$

In this case, we use the `customer-info` information tag to denote messages that contain the kind of information that is affected by the rule, and limit the disclosure of data to the Bank and Government domains, and to their subdomains. In this case, the abstract properties we require are `confidential` and `integrity_verified`. We discuss the mechanisms with which we recognize information characteristics and add information tags in Section 5.3.

Since this is an abstract rule, it can be implemented in several ways. A first option is adopting asymmetric cryptography: for example, when a message is sent from the Bank to the Government, the following rule might be applied in the reference monitors in the Bank domain:

$$\begin{array}{l} \texttt{Bank}\ \{ \\ \quad \texttt{m} : \mathit{message},\ \texttt{x} : \mathit{customer\text{-}info},\ \texttt{m}.\mathit{to} == \texttt{t}\ \mathit{in}\ \mathit{Government} \\ \quad \rightarrow \texttt{x}\ \mathit{is}\ \texttt{asym\_encrypted}(t.P_k) \\ \} \end{array} \tag{3}$$

In this case, `asym_encrypted` is a *concrete* rule applying to all the messages that are sent to any recipient in the Government domain (*i.e.*, whose `to` field is within a policy domain contained in `Government`). This is a concrete rule because it dictates the specific mechanism to use in order to obtain the required property, which is confidentiality.

Such a rule has to be accompanied by other rules: the companion rule enforcing decryption when messages are received in the Government, and a set of analogous rules for messages sent from the Government to the Bank which are tagged with `customer-info` as well.

Other concrete implementations of the same abstract requirement are possible. For example, this can be done with a symmetric cryptography implementation using a shared key:

$$\begin{array}{l} \texttt{Bank}\ \{ \\ \quad \texttt{m} : \mathit{message},\ \texttt{x} : \mathit{customer\text{-}info},\ \texttt{m}.\mathit{to} == \texttt{t}\ \mathit{in}\ \mathit{Government} \\ \quad \rightarrow \texttt{x}\ \mathit{is}\ \texttt{sym\_encrypted}(\mathit{Bank.shared\_key}) \\ \} \end{array} \tag{4}$$

The above concrete policy rule can implement the required abstract property; however, there are cases for which such a solution would not be acceptable: for example, if the abstract property of non-repudiability were requested, it would not be achievable with only this mechanism.

More elaborate scenarios are conceivable: for example, if a reference monitor (say, on a mobile device representing a subdomain of `Bank`) is considered not trusted enough to hold a system-wide shared key - like in the example before - and not powerful enough to process asymmetric encryption, multi-step protocols can be envisaged. In this case, for example, the reference monitor on the mobile device can use a shared key to use symmetric encryption with the `Bank` reference monitor, which can then re-encrypt the messages towards the intended recipient with asymmetric encryption. Such a policy is within the expressive capabilities of HiPoLDS, and can be expressed as follows.

$$
\begin{array}{l}
\texttt{MobileDevice}\ \{ \\
\quad \texttt{m} : \mathit{message},\ \texttt{x} : \mathit{customer\text{-}info},\ \texttt{m}.\mathit{to} == \texttt{t}\ \mathit{in}\ \mathit{Government} \\
\quad \rightarrow \texttt{x}\ \mathit{is}\ \texttt{sym\_encrypted}(\mathit{MobileDevice.shared\_key}), m : \mathit{step}1\_\mathit{applied} \\
\} \\
\texttt{Bank}\ \{ \\
\quad \texttt{m} : \mathit{step}1\_\mathit{applied}\ \rightarrow \texttt{m}\ \mathit{is}\ \texttt{sym\_decrypted}(\mathit{MobileDevice.shared\_key}), \\
\quad \texttt{m}\ \mathit{is}\ \texttt{asym\_encrypted}(m.\mathit{to}.P_k), \\
\}
\end{array}
\tag{5}
$$

In this case, the `step1_applied` information tag is used to mark the first processing step where it is applied; processing will further continue at the `Bank` reference monitor. As before, further matching rules will decrypt messages at the recipient, and deal with sending messages in the opposite direction.

We consider the ability of expressing both abstract and concrete rules as a key feature of HiPoLDS; in Section 5.1 we discuss our plans for deriving or verifying concrete policies based on abstract ones.

4.2 Roles and Policy Domains

It is worth noting that rules based on roles can be expressed via HiPoLDS. Indeed, roles can be expressed by assigning policy domain attributes to policy domains that represent individuals. The following (abstract) rule states that all messages tagged as `classified` should remain confidential between managers:

$$\texttt{m} : \mathit{classified} \rightarrow \texttt{m}\ \mathit{is}\ \texttt{confidential}(\mathit{manager})$$

In this case, we remind that `manager` is a domain attribute, and this rule would be equivalent to enumerating all policy domains with the `manager` attribute. Using domain attributes in this way helps maintainability and avoids repeating the same rule for different domains. The rule can be implemented using concrete rules similar to what we have seen in Section 4.1.

Let us furthermore suppose that we want to avoid sending classified messages to mobile devices, even if they are owned by a manager (*i.e.*, they are subdomains of a domain with `manager` attribute). Such a rule writes as

$$\texttt{m} : \mathit{classified} \rightarrow \texttt{m}\ \mathit{is}\ \texttt{filtered}(\mathit{mobile})$$

In this case, `filtered` is a new property requiring that messages should not arrive to the listed domains. Again, `mobile` is a domain attribute and using it is equivalent to listing all domains tagged as `mobile`.

To enforce this rule, reference monitors in a `manager` domain with `mobile` subdomains should enforce the filtering. This kind of rule is applicable because correctly-behaving reference monitors communicate with each other only through the hierarchical channels as shown in Figure 1 on page V, so the parent node is the only point from which information can reach a domain. It is exactly because, in this case, `mobile` domains are feared to not behave correctly (*e.g.*, have side communication channels) that confidential information is filtered before reaching them.

Here, we reinforce the fact that each reference monitor is as trusted as the domain it is in, and such trust is non-monotonic. In fact, the only reference monitors that handle confidential information unencrypted are those in the domains that have access to it. We point out that this might mean that reference monitors at high levels in the hierarchy might not have any concrete rule to apply – this means that they can effectively be removed. In particular, the top-level global reference monitor could be complex to deploy and to implement, and concrete rules that do not need it could be advisable.

5 From Specification To Enforcement

We focused on describing the design design of the HiPoLDS language; we now turn our attention towards the implementation of an enforcement architecture. We outline the policy refinement process, presenting the main alternatives that will guide the next steps in this research, and then discuss the requirements for the correct implementation of reference monitors.

5.1 From Abstract to Concrete Policies

We describe below the three main approaches for deriving concrete policies from abstract ones.

Fully automated refinement Addressing the semantic gap from abstract to concrete policies: it is necessary to create a translation framework linking the abstract concepts of HiPoLDS policies to lower level system concepts. Assuming that the mapping is correct and reliable, the automated concrete policy generation would produce possible overhead such as for instance, encrypting a communication channel more than once, as soon as the policies governing a domain hierarchy may require confidentiality under overlapping conditions. Nevertheless the major difficulty in such an approach is to create the refinement process itself, by identifying decidable classes of HiPoLDS policies and establishing the correctness of the generated concrete policies considering the formal semantics at the abstract and concrete levels.

Partially automated refinement Since processing abstract policies in HiPoLDS would involve a high number of assumptions about the domain in question with a large number of semantic relationships among the tags, attributes, roles and the domain hierarchy, one can imagine an automated tool would be able to produce a partial concrete policy as an output that would need to be manually edited or corrected by a security expert. As soon as basic considerations about enforcement could be reasonably handled by the automated refinement process, the effort of the security expert could be reduced. However, significant research and development effort would be necessary, probably as much as for the fully automated refinement alternative.

Manual refinement with tool support A more realistic approach would be to rely on the expertise of the security administrator to manually refine abstract policies into concrete ones, using their knowledge of the domain topology and its components. The process can be supported by tools to check the abstract and concrete policies, such as model checkers, which can work without needing much intervention. For instance, the ASLAN++ language and system [18] allows to model distributed systems communication protocols for the verification of security goals.

5.2 Firewall-Based Realization

A common solution to enforce security policies in relying on firewall or network filters. The term *firewall* is commonly associated to packet filter devices operating up to the layer 4 of the ISO/OSI model. The task of analyzing and filtering higher layer protocols (such as HTTP) is instead entrusted to the so-called Application-Level Gateways (ALG). ALGs can, for example, protect against malformed requests or they can be used to limit the access to certain resources.

The same concepts have also been extended to the SOAP protocol. Web-Server firewalling has been studied by academia [12], [4], industry [22], and standard bodies [24]. In fact, most of the vendor in the network security area already propose solutions tailored for the protection of web-services environment (e.g., Citrix Netscaler [17] or CISCO XML Gateway [5]). These firewalls are essentially application gateways that inspect the payload. This approach is less appropriate when payloads are encrypted as the encrypted content necessarily escapes the firewall analysis.

5.3 Introducing Information Tags

The information tags introduced in Section 3.4 are central to our approach to describe security metadata and categorize information. Reference monitors need these tags to correctly enforce HiPoLDS policies.

Information tags can be created in two possible ways: first, as metadata that annotate the processing done on messages by reference monitors; second, by examining and annotating the content of exchanged messages, leading to tags such as `customer-info` in the examples of Section 4.1. The first case is supported in the language by requiring the presence of a tag in the right hand of

a rule, as done for `step1_applied` in rule 5 on page XII; the second depends on the particular implementation of monitored services, and requires inspecting the data which is present in messages and potentially also obtained through cross-layer analysis. The details of how such an inspection should work would depend on a choice of the particular message exchange protocol taken into consideration.

6 Conclusion

In this paper, we introduced the design of and main implementation directions of a new security policy language, HiPoLDS, intended for specifying security in complex distributed systems typically encountered in service-oriented architectures. We believe that, on the one hand, HiPoLDS is expressive enough to describe tersely several real-world policies; at the same time, the complexity of implementing HiPoLDS is manageable.

In future work, we plan to expand on this work with a more formal and concrete definition of the language. Other problems would require further work:

- Reasoning on the relationships between abstract and concrete policies. We can think of inference mechanisms to verify whether concrete policies implement the abstract required policies. Such mechanisms should report inconsistencies between two representation levels or detect non-enforceable rules.
- Weaving concrete policies in the domain hierarchy. Monitors have to be consistently distributed along the hierarchy in order to reliably support the HiPoLDS metaphor of watching the system borders. It is also necessary to provide correct information about the environment, originated from heterogeneous sources and from different architectural layers. We believe that distributed aspect-oriented languages can support the extraction and interception of sensible information that need to be provided to the reference monitors, through the implementation of inlined reference monitors.

References

1. Bauer, L., Ligatti, J., Walker, D.: A language and system for composing security policies. Tech. Rep. TR-699-04, Princeton University (2004)
2. Becker, M.Y., Fournet, C., Gordon, A.D.: SecPAL: Design and semantics of a decentralized authorization language. J. of Computer Security 18(4), 619–665 (2010)
3. Bonatti, P.A., di Vimercati, S.D.C., Samarati, P.: An algebra for composing access control policies. ACM Trans. Inf. Syst. Secur. 5(1), 1–35 (2002)
4. Bunge, R., Chung, S., D., B.E.P., McLane: An operational framework for service oriented architecture network security. In: Proc. HICCS. p. 312 (2008)
5. CISCO ACE XML Gateway. `http://www.cisco.com/en/US/products/ps7314/index.html` (2010)
6. Cuppens, F., Cuppens-Boulahia, N., Ramard, T.: Availability enforcement by obligations and aspects identification. In: Proc. ARES. pp. 229–239 (2006)
7. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder policy specification language. Policies for Distributed Systems and Networks pp. 18–38 (2001)

8. Dantas, D.S., Walker, D.: Harmless advice. In: Morrisett, J.G., Jones, S.L.P. (eds.) POPL. pp. 383–396. ACM (2006)
9. Dell'Amico, M., Idrees, M.S., Roudier, Y., de Oliveira, A.S., Serme, G., Harel, G.: Language definition for security specifications. Deliverable D2.2, The CESSA project (May 2011), http://cessa.gforge.inria.fr/lib/exe/fetch.php?media=publications:d2-2.pdf
10. Douence, R., Grall, H., Mejía, I., Royer, J.C., Südhold, M., Idrees, M.S., Roudier, Y., Leroux, J., Rivard, F., Pazzaglia, J., Serme, G.: Survey and requirements analysis. Deliverable D1.1, The CESSA project (Jun 2010), http://cessa.gforge.inria.fr/lib/exe/fetch.php?media=publications:d1-1.pdf
11. Dougherty, D.J., Kirchner, C., Kirchner, H., de Oliveira, A.S.: Modular access control via strategic rewriting. In: Proc. of ESORICS (2007)
12. Gruschka, N., Luttenberger, N.: Protecting web services from DOS attacks by SOAP message validation. Security and Privacy in Dynamic Environments (2006)
13. Idrees, M.S., Serme, G., Roudier, Y., Oliveira, A.S.D., Graal, H., Sudholt, M.: Evolving security requirements in multi-layered service-oriented-architectures. In: Proc. SETOP (2011)
14. Li, J.X., Li, B., Li, L., Che, T.S.: A policy language for adaptive web services security framework. ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing 1, 261–266 (2007)
15. Moses, T. (ed.): extensible access control markup language (xacml) version 2.0. Tech. rep., OASIS Standard (2005)
16. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering and Methodology 9(4), 410–442 (2000)
17. CITRIX NetScaler. http://www.citrix.com/english/ps2/products/product.asp?contentid=21679 (2010)
18. von Oheimb, D., Mödersheim, S.: ASLan++ - a formal security specification language for distributed systems. In: Prof. FMCO (2010)
19. de Oliveira, A.S., Wang, E.K., Kirchner, C., Kirchner, H.: Weaving rewrite-based access control policies. In: Ning, P., Atluri, V., Gligor, V.D., Mantel, H. (eds.) FMSE. pp. 71–80. ACM (2007)
20. Ribeiro, C., Ferreira, P.: A policy-oriented language for expressing security specifications. International Journal of Network Security 5(3), 299–316 (2007)
21. Ribeiro, C., Zuquete, A., Ferreira, P., Guedes, P.: SPL: An access control language for security policies with complex constraints. In: Proc. of NDSS (2001)
22. Room, S.I.I.R.: XML Firewall Architecture and Best Practices for Configuration and Auditing. http://www.sans.org/reading_room/whitepapers/firewalls/xml-firewall-architecture-practices-configuration-auditing_1766 (2007)
23. Serban, C., Zhang, W., Minsky, N.: A decentralized mechanism for application level monitoring of distributed systems. In: Collaborative Computing: Networking, Applications and Worksharing, 2009. CollaborateCom 2009. 5th International Conference on. pp. 1–10. IEEE (2009)
24. Singhal, A., Winograd, T., Scarfone, K.: Guide to Secure Web Services. NIST Publication
25. Song, E., Reddy, R., France, R.B., Ray, I., Georg, G., Alexander, R.: Verifiable composition of access control and application features. In: Ferrari, E., Ahn, G.J. (eds.) SACMAT. pp. 120–129. ACM (2005)