A General Language for Expressing Security Policies for Enforcement

James A. Hoagland, Raju Pandey, Karl N. Levitt[1]
Department of Computer Science
University of California, Davis
{hoagland,pandey,levitt}@cs.ucdavis.edu

Abstract

There is a compelling need for organizations to create and make known their security policies, for the benefit of both employees and the configuration of security system components. Currently, security policies are stated in English. We believe that the primary impediment to more specific and mechanizable security policies is the unavailability of a formal language that supports the creation, analysis, understanding, and mechanization of real security policies. Current security models are too abstract and not representative of particular security needs of most organizations. We present LaSCO, the Language for Security Constraints on Objects, a formal policy language based on directed graphs for stating policies on many kinds of systems. LaSCO can express standard safety models for systems as well as custom policies for a site, a key capability in making it useful. We have implemented LaSCO so as to facilitate stating security constraints for, and mechanically enforcing constraints on, Java programs. We find LaSCO suitable for expressing and enforcing policies on distributed object systems.

1 Introduction

There is a compelling need for organizations to create and make known their security policies. Through the policies, the expectations of employees and the system components that enforce security are clear. Currently, security policies are stated in English, are usually vague, and the translation to operating procedures or mechanized enforcement components is manual. We believe that the primary impediment to more specific and mechanizable security policies is the unavailability of a formal language that supports the creation, analysis, understanding, and mechanization of real security policies.

Despite the absence of such a language, there is increasing interest in security policies and languages to express them. There are perhaps a dozen security models that have been widely studied and serve as benchmarks for security implementations and languages – including ours. However, we believe that these models are too abstract and not representative of particular security needs of most organizations.

In this paper, we present LaSCO (Language for Security Constraints on Objects), a language to express security policies involving access to distributed objects, where the policies typically involve safety: what accesses are not allowed. The language is strongly typed and extensible in that new objects and events can be declared. Different from approaches to specifying policy based on the access matrix model, LaSCO permits policies based on the history of prior accesses, the states of objects, and ordering relations among accesses. Specified accesses can be concurrent or specified to satisfy partial orderings or to occur at specific times. As distinct from some policy approaches, LaSCO has a diverse range of systems[2] to which it may be applied. It may be applied to any system that can be modeled as consisting of events that interact through events. This includes program execution, file systems, operating systems, and networks of hosts. LaSCO supports what we call policy engineering. Policies are specified as graphs (nodes representing objects and edges representing events) thus providing such visual metaphors as the clustering of activities as cliques and the ordering of accesses. LaSCO policies are translatable into formal logic, thus permitting reasoning about the completeness and consistency of a policy. Also LaSCO policies are executable in that they are directly translatable into executable assertions that are, currently, attached to Java programs. A primary application of the current implementation of LaSCO is to detect policy violations associated with Java code, but in principle the approach can be used to detect policy violations associated with any combination of system and application programs.

A policy contains two types of predicates that are assigned to objects and methods invocations. Domain predicates identify assumptions on the system state; requirement predicates identify conditions that must be true if the assumptions

1. We gratefully acknowledge the support for this research by DARPA under contract XXX-XXX and the Intel Research Council under grant XXX-XXX.

2. We use "system" generally in this paper, to include any computational entity.

are satisfied. In essence, a policy is a rule of the form domain ⇒ requirement. LaSCO provides numerous built-in theories to assist in policy formulation, including propositional logic, a restricted form of predicate logic and set theory, and temporal logic. In this paper, we discuss policy composition using conjunction and disjunction semantics. Using a form of denotational forms, the operational semantics of LaSCO policy primitives has been formalized [1].

As we discuss, LaSCO can express all of the standard safety models: multi-level security, discretionary access, Clark-Wilson, role-based access control, and Chinese Wall. These are expressed generically, and can be instantiated to apply to any system. LaSCO can also be used to describe acceptable actions of Unix privileged programs, namely the basis for specification-based intrusion detection – see Calvin Ko's UC Davis thesis [11], Sekar's intrusion detection system [18].

Beyond the standard security models, LaSCO is well-suited for customized policies involving access control, for example: A student is not to have access to a failing grade until a teaching assistant not affiliated with the class has checked the grade followed by approval for release by the department chair.

A Java implementation of LaSCO demonstrates its effectiveness in the enforcement of policies associated with the execution of object-oriented programs. The implementation consists of: Schema extraction (to identify the objects and method invocations associated with a Java program), Policy construction (whereby the user creates a policy for the extracted schema, and displays it through a Perl/Tk interface), and Compilation (which embeds the policy into the Java implementation).

We have discovered that LaSCO is well-suited for the expression of security policies for accesses to distributed objects. In this environment, not all the events and objects needed for detecting a subset of the system history in which a policy domain matches may be available in one location. LaSCO supports distributing parts of its domain for separate detection, followed by aggregation. LaSCO, currently, cannot be used to express liveness policies (such as that particular events are obliged to occur).

The body of this paper is organized as follows: Section 2 presents LaSCO and its system basis. We present several examples of LaSCO policies in Section 3. Section 4 presents ways of applying LaSCO to systems in general, Java program and distributed object systems in particular, and discuss our prototype implementation. In Section 2.8 we discuss the expressiveness of the language and Section 5 presents a comparison of our work with related work. Section 6 concludes and discusses future work.

2 LaSCO

Our policy language, LaSCO describes constraints on a system that must hold as a function of the system state. The described state accounts for events that have occurred. We use a ***policy graph*** to represent both the situation under which a policy applies (the ***domain***) and the constraint that must hold for the policy to be upheld (the ***requirement***). Thus a LaSCO policy states is assertions that indicates if the system is in a specific state, the events and objects of the system must satisfy a set of properties. See Figure 1 for an example. This policy graph depicts the simple security property of Bell-LaPadula [3]:

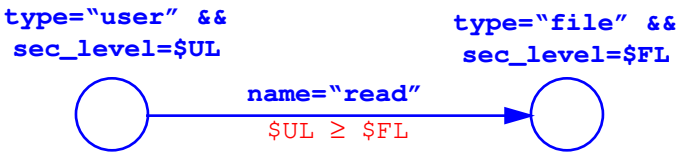


Figure 1. Policy graph for the simple security property

if a user is reading a file, the security level of the user's clearance must be at least as great as that of the file's classification.

2.1 The system basis for LaSCO

Before we describe LaSCO in detail, we consider the system to which it is applied.

Policies are in effect over a system. We model a system as a set of objects and a set of events. An object is a system entity that has a state. An event can be any type of access or communication between a pair of objects at a given time instance. A ***system history*** is viewed across a range of time) can thus be seen as a sequence of events occurring between a set of objects. Due to the generality of this model, we can model a variety of systems including programs, operating systems, file systems, and networks.

We present an example system history in Figure 2. There are four objects and three events. Each object has a set of

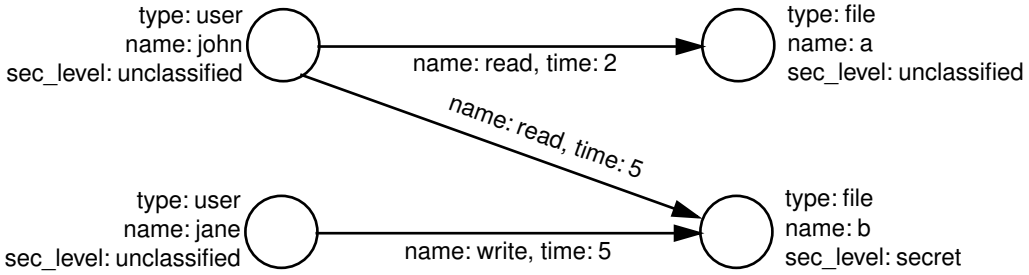


Figure 2. Example system depicted as a graph. The nodes represent objects and the edges events.

attribute-and-value pairs that represents its current (security-relevant) state. For example, the user objects have three attributes each. The *type* attribute denotes the type of the object, the value of the *name* attribute is the name of the user, and *sec_level* represents the clearance level of the user. Similarly, each event has a set of parameter and value pairs that denote the details of its invocation. In this simple example, each event has only two parameters: *name*, whose value is the name of the event and *time*, whose value is the time of its execution. The policy associated with the system, as we discuss below, specifies if the events are allowed to occur.

The parameters of events are static. However the attribute values on objects may vary over time, thought the set of attribute names present is invariant. There is a constant unique 'id' attribute associated with each object.

We assume that events and changes to object attribute values occur at discrete time steps and that multiple events may happen simultaneously. This allows us to view our system history as a sequence of system instances, each representing the system at a particular time. In addition to a set of objects, each system instance contains a set of pending events, events that are intended to execute, but are awaiting approval. It is at this point where policy is considered for a system history. We denote the time of a system instance in each of its events using a 'time' parameter.

2.2 Predicates

Policy graphs are annotated directed graphs. The annotations on a policy graph are termed ***predicates***. There is a ***domain predicate*** (depicted with **`bold text`**) and a ***requirement predicate*** (denoted with `standard text)` for each node and edge in the policy graph. (Nodes and edges without an explicit domain or requirement predicate have a default "True" predicate.) Domain predicates describe the kind of objects or events that are relevant to a part of the policy. Requirement predicates describe what must hold where the domain is satisfied. Although they serve different roles, both predicates are evaluated in the same way. In either case predicates are patterns on the attributes of an object or on the parameters of an event; predicates succeed for objects or events that are described by the pattern.

For the moment, let us consider only predicates without variables, which we will term ***simple predicates***. We will introduce policy variables in Section 2.4. A simple predicate is a boolean expression formed from attribute or parameter names and constants combined by operators from a predefined set of logical, comparison, set, and mathematical operators. Parentheses may be used to nest subexpressions. Predicates are evaluated in the context of attributes or parameters; applying the predicate consists of substituting in the corresponding value for each name and resolving the resulting constant expression to be true or false.[3]

2.3 Domain matching

The domain of a policy (which describes conditions under which the policy applies) is a set of domain predicates, nodes, and edges. To interpret a LaSCO policy, we identify the locations in the system history where the domain matches, and then check the policy requirement for each match. Let us now consider the process of matching the domain of a policy to a subset of the system history. We address this for simple domain predicates.

The domain pattern is satisfied when each node and edge in the policy graph is satisfied by a subset of a system history. This would consist of an object for each node and an event for each edge, in an one-to-one association. In the simple predicate case, this mapping between nodes and objects and edges and events constitutes what we term a ***policy to system match*** (***match*** for short). As the domain may apply in several ways in the system history or not at all, applying the domain to the system history produces a set of these matches.

3. In the case that an attribute or parameter name appears in a predicate but not in the object or event, the most immediate boolean expression in which the name appears evaluates to false, regardless of any other part of that expression. This implies that, unless that boolean expression is within a disjunct expression, the predicate will not be satisfied by the object or event.

Consider an policy edge *E* that we are considering as matching a system event *e* that occurs at time *t*. For *E* to match *e*, this criteria must be met:

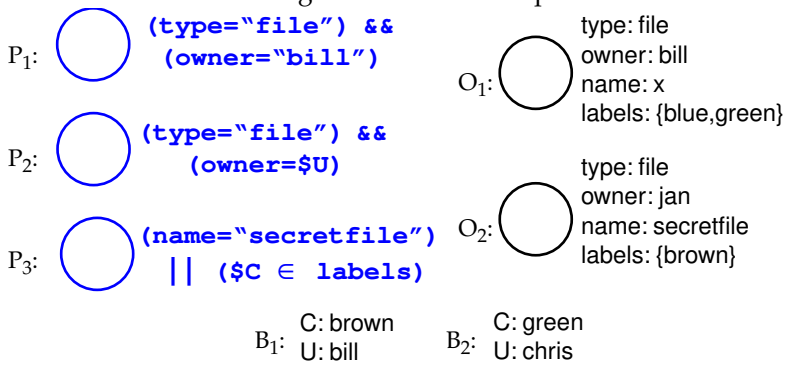
1. the domain of *E* matches *e*
2. the domain of *src(E)* matches *src(e)* at time *t*
3. the domain of *dest(E)* matches *dest(e)* at time *t*
4. *src(e)* is what matches *src(E)* for all policy edges involving *src(E)*
5. *dest(e)* is what matches *dest(E)* for all policy edges involving *dest(E)*

Thus each node must match the same object for each edge it is incident with. Isolated policy nodes, those with no incident edges, are easier to match. They can match a system object at any time.

2.4 Variables

LaSCO policies make use of logical policy variables to relate attribute and parameter values associated with different objects and events. Variables may appear as operands in domain and requirement predicates and are denoted by a "$" prefix. The scope of a variable is a single LaSCO policy graph. Within this scope, each variable is bound to exactly one value.

Variable bindings represent a set of policy variables that have a bound value. Predicates are evaluated in the context of a set of variable bindings. We demonstrate predicate evaluation through a simple example. Figure 3 presents several



	O_1	O_2
P_1	satisfied with any variable binding	not satisfiable by any variable binding
P_2	satisfied by B_1 but not B_2	not satisfied by B_1 nor B_2
P_3	satisfied by B_2 but not B_1	satisfied by either B_1 or B_2

Figure 3. Predicate evaluation example. P_1, P_2, and P_3 are policy nodes, O_1 and O_2 are system objects, and B_1 and B_2 are variable bindings. The table depicts which variable bindings satisfy each policy node's predicate when evaluated in the context of the object.

example pattern nodes, system objects, and variable bindings. The table uses these to give examples of predicate evaluation on system objects. For example, P_2 evaluated in the context of O_1 and B_1 yields `("file"="file") && ("bill"="bill")`, which is true, so the predicate is satisfied. However P_3 evaluated in the context of O_1 and B_1 yields `("x"="secretfile") || ("brown" ∈ {"blue","green"})`, which is false, so P_3 is not satisfied.

We now describe two restrictions on predicate contents. Each variable present in a policy must appear in one or more domain predicate subexpressions as <variable>=<value> or <value>=<variable> where <value> is a single value. This subexpression can not be part of a disjunction. This restriction ensures that all variables have a single value for a domain. The second restriction is that node requirement predicates may not contain attributes as operands[4].

2.5 Domain matching with variables

A policy domain is satisfied when all of its nodes and edges can simultaneously be satisfied by a set of variable bindings. Given a domain with variables, a policy to system match, in addition to containing a map between the policy and system history, also contains the set of variable bindings that enable the mapping.

4. The values of an attribute might vary during the different times of the matching events that are incident to the object. This would lead to ambiguity if the attribute is mentioned in a requirement predicate. Thus, we impose this restriction to alleviate this ambiguity. Variables may be bound to an attribute value in the domain and referred to in the requirement, which ensures that the hold a single value.

Let us now consider domain satisfaction for the policy in Figure 1 in the context of the example system history depicted in Figure 2. In Figure 4, we depict the application of the policy domain to the system history in by overlaying the

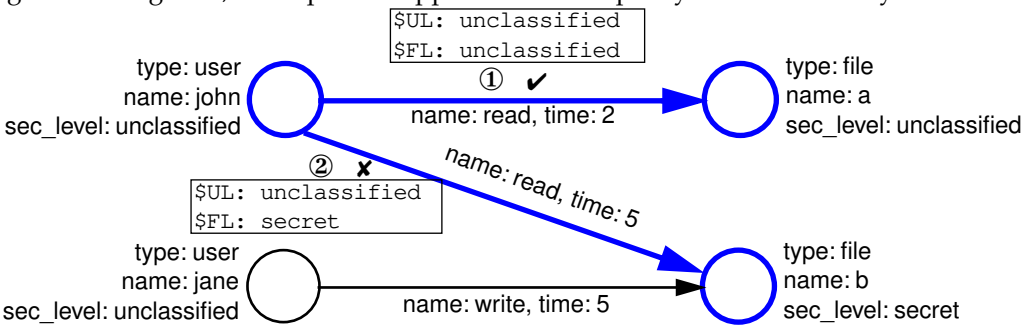


Figure 4. Depiction of simple security property applied to the example system. The two places where the domain applies is noted along with the necessary variable bindings.

policy graph on a graph of the system history and noting the necessary variable binding. When the domain of the policy is applied to the system history, the policy graph matches in two locations, indicated by the ① and the ② and the thick lines. Note that the domain did not match for the third event, meaning it is not covered by the policy, because our policy only covers read accesses.

2.6 Requirement checking

The policy requirement is the set of requirement predicates in a policy graph. A requirement predicate is evaluated against a match. If each of the requirement predicates evaluates to true for a match, the policy has been upheld, otherwise the policy has been violated by the match. Edge requirement predicates are evaluated in the context of the parameters of the event that matched the edge in the match for the variable bindings in a match. A node requirement predicate is evaluated in the context of the variable bindings from the match. The policy does not dictate a particular response to a policy violation.

Consider the matches in Figure 4 noted earlier. The policy requirement (`$UL ≥ $FL`) is satisfied by ① but not ②. ① succeeds because `"unclassified" ≥ "unclassified"` is true but ② does not because `"unclassified" ≥ "secret"` is false. Thus, ① is allowed by policy but ② is not.

2.7 Policy composition and operations

We have defined some formal operations on policies in [1], of which we summarize two here: conjunction of policies and disjunction of policies. More than one policy might be applied to a given system (and thus on their histories). The usual intention when applying these together is that each of these be enforced as if no other policies were being applied. That is, each of the individual policies must be satisfied for the set of policies to be satisfied. We term this form of composition *conjunction*.

An alternate semantics that may sometimes be desired is what we term *disjunction*. Disjunction between a pair of policies implies that in cases where both policies' domains apply, only one of the requirements need be satisfied. When only one of the policies' domains applies, the requirement for that policy must be satisfied. No requirements are in place by the two policies if neither of the domains apply.

2.8 LaSCO expressiveness

LaSCO has a far more general system constraint model than traditional access control models. This enables the stated policies to be closer to the intention that would otherwise be possible and to be applied to non-traditional environments such as distributed object systems. Using LaSCO, one can state policies that restrict the actions on a system, possibly based on what has occurred previously. Policies can also impose constraints on the state of an object. The system model employed by LaSCO is simple, very flexible, and we find it adequate for modeling security aspects of systems that relate to access control.

Some constraint policies cannot be stated in the current LaSCO in part because we wish to keep the initial release of language simple. LaSCO cannot state system liveness policies, including those in the form of obligation. An example of this is the policy that employees must execute orders given by their supervisor. A technical limitation that LaSCO has is that it cannot state a policy that refers to an object having to be in two or more distinct states. (We have not seen a natural instance of this sort of policy though.) Also in LaSCO, it is not possible to express the fact that in order for the policy to be

applicable certain events should not occur. An example of this is policies that refer to events that occur without other events having occurred. We have considered several extensions to LaSCO to address these limitations [1].

3 Examples

This section presents several examples of the use of LaSCO.

3.1 Single-event policies

Policies involving just a single event and its associated objects correspond to a LaSCO policy graph containing a single edge and adjacent nodes. One example of this was the simple security property in Figure 1. Access control lists [7] and access control matrices [14] can be represented in LaSCO with one or more single-edge policies. For example, Figure 5

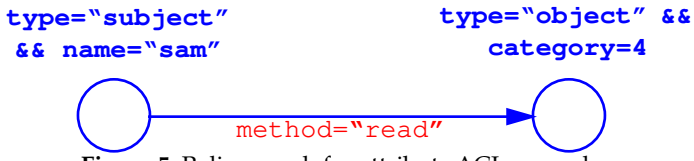


Figure 5. Policy graph for attribute ACL example

depicts the following policy: the only method the subject "sam" is permitted when accessing an object in *category* 4 is read.

Role-based access control (RBAC) [16] can also be represented in LasCO. In modeling a RBAC system, one can denote the roles a user currently has active by a *roles* set attribute on objects that are of the type subject. Figure 6 denotes the

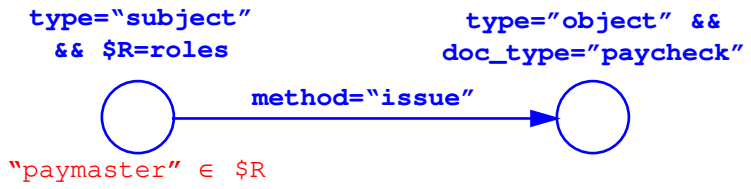


Figure 6. Policy graph for payroll RBAC example

RBAC policy that only subjects that have the "paymaster" role, can issue a object of document type paycheck.

3.2 History-based policies

Policies where we need to check a requirement involving multiple events, represented in LaSCO as a graph with multiple edges. Consider the Chinese Wall policy [4]. The idea behind the Chinese Wall policy is to prevent conflict of interest situations by consultants that may be employed by several parties with competing interests. The policy specifies separation of interests by forbidding any consultant from accessing data from different parties where the parties are in the same conflict of interest class. The policy is depicted in LaSCO by a node with two edges originating from it as shown in Figure 7. The middle node represents a "consultant" whose accesses are limited by the Chinese wall policy. The edges from

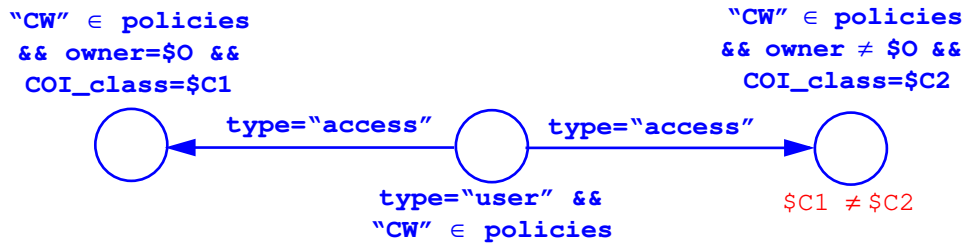


Figure 7. Policy graph for the Chinese Wall policy

the consultant node represent accesses to sensitive objects with different owners that are subject to the Chinese Wall policy. The constraint is that the owners of these objects cannot be in the same conflict of interest class, which we assume to be denoted in the system in the attribute "COI_class".

Separation of duty policies, policies requiring different peoples' involvement in some transaction, can be depicted in LaSCO. An example is shown in Figure 8. This depicts a policy where there is a separate function for requesting policies

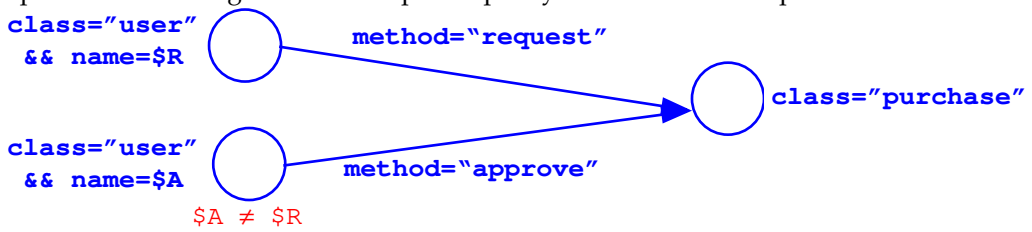


Figure 8. Policy graph for purchase request and approval separation of duty

and having them approved. The policy states the restriction that the "request" user must be different than the "approve" user. Time-ordered access restrictions can be represented similarly.

LaSCO can be used to impose a limit on the number of accesses of a particular type. As an example, consider the LaSCO policy depicted in Figure 9, where we show a policy that might be desired on a program that serves images from a

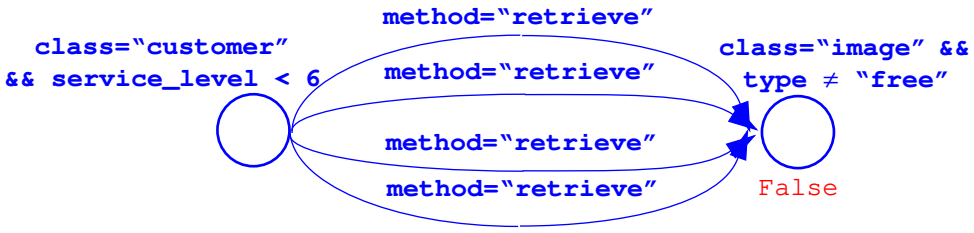


Figure 9. Policy graph for image retrieval quantity restriction

database. The restriction is that, for a customer with assigned service level less than 6 and for images that are not free, the image cannot be retrieved by the customer more than three times. For this policy the domain matches on the fourth retrieve and there is no way to uphold the policy at that point since the false requirement predicate evaluates to false always. Thus a fourth access of this kind is always denied. (As syntactic sugar, we have considered an extension to LaSCO that adds an iteration count to edges.)

3.3 Object state restriction policies

LaSCO states restrictions just on object state in the form of policy graphs that contain one or more isolated nodes, policy nodes without incident edges. We present an example of this in Figure 9. The policy in this figure states that the "/

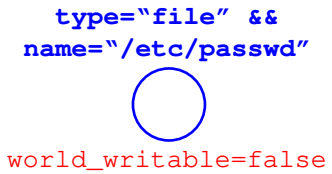


Figure 10. Policy graph for password file restriction

etc/passwd" file should never be world writable.

4 Applying and implementing LaSCO

In this section, we discuss the application and enforcement of LaSCO on systems and our implementation of LaSCO. We begin by discussing some concepts found in applying LaSCO to system histories. This is used in the remainder of the section.

Recall that when the domain of a LaSCO policy is relevant to a portion of the system history, it forms a match. On real systems, when we wish to detect policy matches, we notice some parts of the match before other parts. In fact, some parts may not have occurred when first checking. Given this, a useful abstraction can be a ***partial match***. Partial matches for a policy possess some subset of the semantic elements of the policy filled in (matched) and some conditions on policy variable values. These ***semantic elements*** are each policy edge (and its incident nodes) and each isolated policy node. ***Variable conditions***, an concept from LaSCO's formal semantic definition [1], are the logical conditions for variables under which a predicate or a set of predicates is satisfied. Variable conditions represent the requirements on the value of variables that enable a partial match.

Towards building a complete match, two partial matches may be merged if they agree on which subsets of the system history match which subgraphs of the policy and if their variable conditions are consistent. That is, they don't contradict on what the value of a variable should be. A more complete partial match is the result of merging partial matches. This has the union of the maps between policy and system history. The new variable conditions is the conjunction of the variable conditions of the original partial matches. Note that by the point at which a partial match becomes complete, its variable conditions can be extracted into a single set of variable bindings, due to the variable assignment restriction from Section 2.4.

4.1 An architecture for implementing LaSCO policies

As LaSCO is independent of any specific system or enforcement mechanism, we now consider how to apply LaSCO to a particular system. Our approach is to use a generic policy interpretation engine and an interface layer for each particular system.

A generic policy engine is capable of detecting policy violations on any system that meets the LaSCO system model. At an abstract level, a policy engine accepts as input the appropriate set of LaSCO policies. In addition it receives events and object state changes. The output of the policy engine is policy to system matches that violate the policy.

The interface layer is customized for specific systems. This layer may be in-line code, a wrapper around entities of a system, or potentially something in-between. Regardless of how it is done, the interface layer monitors the system and passes the relevant events and object state changes to the policy engine. Another aspect of the interface layer determines which policies should be in effect on the system and notifies the policy engine of this.

4.2 Our policy engine

We have implemented a generic LaSCO policy interpretation engine in Perl, whose operation we summarize here. A LaSCO policy is loaded from a file to create an instance of a *LaSCO* class to serve as a policy interpreter for the policy. The *SystemGraph* class represents the system history. It is created by an abstraction layer.

A *LaSCO* class method accepts a *SystemGraph* instance and finds all violations of a policy. It returns all system to policy matches in which the policy is violated to find policy violations of a policy. The algorithm employed here initially finds all partial matches to the semantic elements of the policy. These partial matches are then merged to form all the complete matches using a depth first search. The order in which these partial matches are considered for complete matches are carefully considered to greatly reduce the expected number of branches followed. A future *LaSCO* class method would allow discovery of only those policy violations involving certain parts of the *SystemGraph*. This would use a similar algorithm to that of the above method and can be used to locate violations in a recent subset of the system history.

In the policy engine, predicates are evaluated by substituting attribute values for their names and then simplified by the optimization technique of constant folding. The result is variable conditions for the node or edge match. Variables with uniquely determined values are kept separated in variable bindings from the rest of the variable conditions, so they may be used to test for unifiablity.

4.3 Our implementation for Java

We now describe our implementation towards applying LaSCO to Java programs. In Figure 11, we show the steps

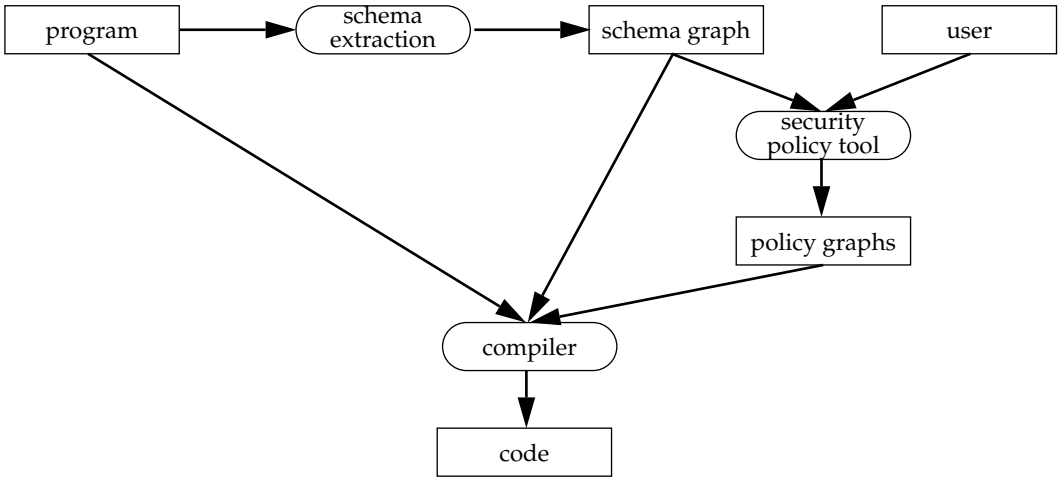


Figure 11. Specifying and enforcing LaSCO policies on Java programs.

that support a user as he writes a security policy for a Java program and sees it through linking to a program:

- Schema Extraction: Given a Java program, a schema extraction tool constructs a program schema graph. Nodes of the schema graph denote class definitions of the program, whereas edges represent method invocations or object instantiations identified in the program source. Our implementation can save the extracted schema in a file in either schema format or in schema graph format.
- Policy construction: A graphical user interface allows LaSCO policies to be edited, created and saved, with schema graphs from a Java program available to facilitate this process. Figure 12 shows a snapshot of the editor and

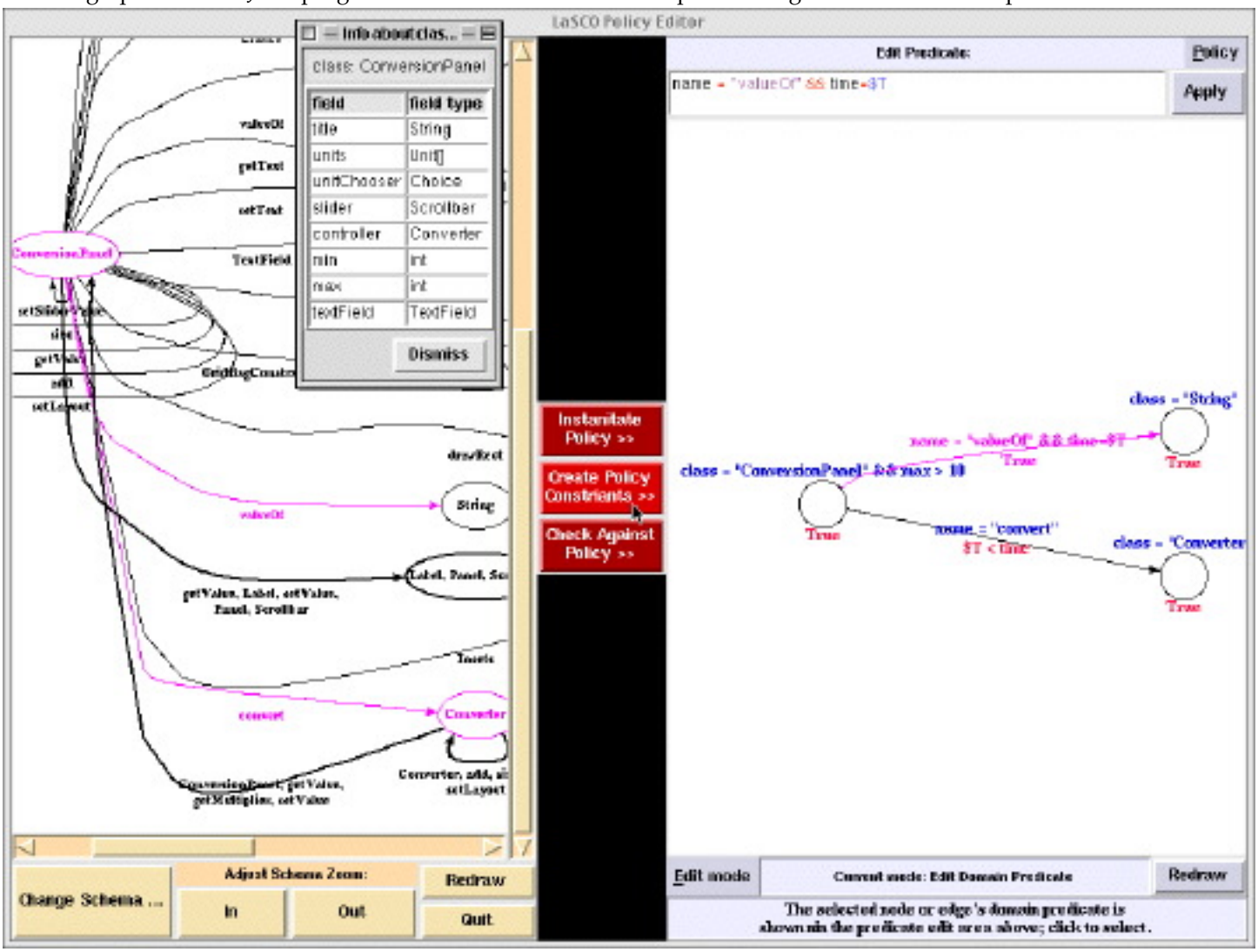


Figure 12. Screen snapshot of the LaSCO policy editor interface.

Figure 13 lists some of the function available through the editor. Generic policies may be loaded and created using

Schema viewing functions:

- load program schema from a Java source file, schema file, or schema graph file
- view class information for a node
- view method information for an edge
- adjust view of schema by zooming in and out and scrolling
- compress selected nodes into a representative supernode
- compress the edges between a pair of nodes into a representative superedge

Policy creation functions:

- create an empty policy
- load or insert a (generic) policy from a file
- save the current (generic) policy
- add a node or edge to the policy
- delete a node or edge from the policy
- edit a domain or requirement predicate
- check syntax of and type use in predicates

Schema and policy integrated functions:

- Instantiate part of a generic policy with part of the schema graph
- Insert a policy domain that matches for exactly the classes and method invocations in part of the schema graph
- Check the policy against (part of) the schema graph to find matches and violations, to the extent statically possible

Figure 13. Some functions available through the policy editor interface.

the interface. These are LaSCO policies with certain portions of predicates that get filled in when applied to a part of a schema graph. This helps facilitate policy construction. Through the interface, a user may statically evaluate a policy against a schema graph. This is mechanized through the generic policy engine in the previous section. For

this, the system history graph is constructed from the schema graph. The policy interface is implemented in about 7200 lines of Perl/Tk code, excluding the schema extraction functionality.

- Compilation: A compiler takes the program, access control constraints represented in policy graphs, and a program schema graph and generates code for both implementing the program and for enforcing the access constraints. This has not yet been implemented but may involve wrapping method calls or the use of JavaBeans.

4.4 Applying LaSCO to distributed object systems

On distributed object systems, the information needed to make policy decisions (object state and events) may not be available from any particular location. Thus, policy enforcement needs to be decentralized. The semantics of LaSCO lends it to this, as the domain of a policy may be divided into its semantic elements. The following procedure may be used to find LaSCO policy violations in a distributed object system:

- Decomposition: Create a ***policy piece*** for each edge and for each isolated node. These pieces include the domain and requirement predicate for each node and edge in them.
- Distribution: These policy pieces are then distributed to all locations in the system in which it may match. Some intelligence might be used in this, if the kind of object or event described is never seen by part of the system, then there is no use looking for it there. If two locations in the system are guaranteed to see the same event or object, then there is no need to look for it in both places.
- Monitoring: The system is monitored in the various locations containing policy pieces for matches to each of its policy pieces. Each match to a policy piece is a partial match to the policy. Requirement variable conditions are also noted with the match. These are determined by substituting attribute values for attribute names in requirement predicates.
- Aggregation: The details of this step might be best determined depending on the architecture of the system and the environment in which it is being used. Using some specific protocol, the partial matches are accumulated to form complete matches. The simplest method is to have a central policy engine that forms complete matches, perhaps in a manner similar to that described in Section 4.2. The variable bindings in a complete match are used to evaluate each of the requirement variable conditions. If any of these do not evaluate to be true, then the policy is violated by the match.

5 Comparisons with other work

The Adage architecture [19], developed at the Open Group Research Institute, focuses on creating and deploying security policies stating access control on roles in a distributed environment. The developers argue that security products that a user can not understand will not be used and focus on usability through enabling the user to build policy from pieces that the user understands. We agree with this sentiment and feel LaSCO has something to offer in terms of usability (though no usability study has been performed). We believe LaSCO can express any Adage policy, but offers the additional benefits of application to different kinds of systems, direct linking to an application program, and formal semantics.

The policy language presented in this paper took some inspiration from the Miró work of Heydon, Tygar, Wing, *et. al.* at Carnegie Mellon University [13]. Using their constraint language, one can state allowable access control states and file and group nesting for file systems. They employ visual means, an annotated graph based on Harel's hierarchical graphs [12]. However, Miró can only express allowable states (a snapshot of a dynamic system), whereas LaSCO can state constraints based on events. Also, LaSCO can be applied to systems other than file systems and has simpler semantics.

The traditional decision model of access control in computers has access decisions based on the subject, object, and privilege requested. This is what is found in access matrices [14], access control lists, and capability-based systems (see [7]). Some extensions to this model are have been proposed, such as TAM [17] which introduces safety properties into an access control matrix and BEE [15] which can make reference to global attributes in evaluating an access. The Authorization Specification Language [10] introduces groups and roles in the model and provides rules to express policy beyond a single access control policy. These models can be coerced into providing some ability to express policies for distributed object systems (for a method invocation, call the invoking object the subject, the destination object the object, and the method being invoked the privilege requested). However, this does not allow for decisions based on the context of the system (states of objects and other invocations that are occurring or have occurred) or for policies restricting the state of objects. These kinds of policies seem desirable, and LaSCO can express these. Also, in a distributed object system, one cannot assume a centralized global state, which some of these mechanisms depend on.

Some policy languages aim to express liveness properties such as obligation. For example, Cholvy and Cuppens [5] aim to express all the policies on a site in terms of deontic logic. So as to keep the language simple and readily enforceable, LaSCO at present is focussed on safety. Other policy models and languages such as Goguen and Meseguer [9] and Woo and Lam [20] express policies formally in terms of logic, but provide no guidance as to how to enforce the expressed policies on a system. LaSCO has formal semantics and is readily enforceable.

Deeds, developed by Edjlali, Acharya, and Chaudhary [8], is a history-based access control mechanism for Java whose goal is to mediate accesses to critical resources by mobile code. LaSCO can also be used for this purpose. As we are doing, they insert code into Java programs. However, whereas their basis for access control decisions is the result of dynamically executing Java code provided by the user, our basis is clearly stated policies. We find our approach appealing since it permits conceptual understanding of the access restriction and formal reasoning about the policy.

6 Conclusion and future work

In this paper, we presented a formal policy language based on graphs. LaSCO policies separate into two components. The domain of a policy matches when objects and events in a system's execution are found that match the kind described by domain predicate annotations on the policy graph. When this occurs, the policy requirement is checked. If the requirement is not satisfied by the subset of a system history that matched, the policy has been violated. It may express standard notions of systems safety such as Bell-LaPadula [3] and Chinese Wall [4]. As a result of the flexibility of LaSCO, an application developer or site can create custom policies to fit their needs. In particular, policies that are dependent on system historical context and that restrict the state of objects may be expressed. This customizability promotes security.

This language may be used to express policies for any system that can be modeled as consisting of objects that interact through events. The syntax and semantics of LaSCO lends itself to use in distributed object systems, where portions of a system execution that can match the domain of a policy may need to be observed from several locations.

We have implemented a generic policy interpretation engine in Perl. Using this, one can find the places in a system's execution that violate a set of LaSCO policies. One use of this is our application of LaSCO to Java programs. We have implemented the schema extraction and policy editing tools for implementing LaSCO policies. The policy editor presents a program's schema and a LaSCO policy visually and provides interaction mechanisms between them to serve as an aid in writing policies for the program. We are beginning to implement the policy compiler, which ensures that a Java program observes the appropriate LaSCO policies.

We are also studying enforcing LaSCO by the GrIDS intrusion detection system [6]. To accomplish this, we will translate the policies into a modified version of the GrIDS ruleset language. This will allow policies written to be enforced over a large network of computers and allows the policies to make use of inputs from an intrusion detection system such as the alert level on a host. We will further pursue research in enforcing LaSCO on distributed object system. Another area of further study will be policy composition and operations and finding conflicts between policies.

7 References

[1] Hoagland, James, Raju Pandey, and Karl Levitt, "Security Policy Specification Using a Graphical Approach." Technical report CSE-98-3, The University of California, Davis Department of Computer Science. July 1998.

[2] Anderson, J.P., "Computer Security Technology Planning Study," ESD-TR-73-51, Vols. I and II, USAF Electronic Systems Division, Bedford, Mass., October 1972.

[3] Bell, D.E. and L.J. LaPadula, "Secure Computer Systems: Mathematical Foundations and Model," M74-244, The MITRE Corp., Bedford, Mass., May 1973.

[4] Brewer, D.F.C., and M.J. Nash, "The Chinese Wall Security Policy," In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, Oakland, CA, USA: IEEE Press, 1989.

[5] Cholvy, Laurence and Frederic Cuppens, "Analyzing Consistency of Security Policies." In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. Oakland, CA, USA: IEEE Press, 1997. p. 103-112.

[6] Cheung, Steven, Rick Crawford, et. al., "The Design of GrIDS: A Graph-Based Intrusion Detection System." Technical report CSE-99-02, The University of California, Davis Department of Computer Science. January 1999.

[7] Denning, Dorothy, *Cryptography and Data Security*. Addison-Wesley, Reading, Mass. 1982.

[8] Edjlali, Guy, Anurag Acharya, and Vipin Chaudhary, "History-based Access-control for Mobile Code." In *Proceedings of the Fifth ACM Conference on Computer and Communications Security*. San Francisco, CA, USA. November 1998.

[9] Goguen, J.A. and J. Meseguer, "Security Policies and Security Models." In *Proceedings of the 1982 Symposium on Security and*

Privacy, pp 11-20, 1982.

[10] Jajodia, Sushil, Pierangela Samarati, and V.S. Subrahmanian, "A Logical Language for Expressing Authorizations." In *Proceedings of the 1997 IEEE Symposium on Security and Privacy.* Oakland, CA, USA: IEEE Press, 1997. p. 31-42.

[11] Ko, Calvin, "Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-based Approach", Ph.D. Thesis, University of California, Davis, 1996.

[12] Harel, David, "On Visual Formalisms." *Communications of the ACM*, 31(5):514-530, May 1988.

[13] Heydon, Allan, Mark W. Maimone, J.D Tygar, Jeannette M. Wing, and Amy Moormann Zaremski, "Miró: Visual Specification of Security." In *IEEE Transactions on Software Engineering*, 6(10):1185-1197, October 1990.

[14] Lampson, B.W., "Protection," In *Proceedings of the 5th Symposium on Information Sciences and Systems*, Princeton University, March 1971.

[15] Miller, D.V. and R.W. Baldwin, "Access control by Boolean Expression Evaluation." In *Proceedings Fifth Annual Computer Security Applications Conference*. Tucson, AZ, USA: IEEE Computer Society Press, 1990. p.131-139.

[16] Sandhu, R.S., E.J. Coyne, H.L. Feinstein, and C.E. Youman, "Role-based access control: a multi-dimensional view." In *Proceedings of the 10th Annual Computer Security Applications Conference*, Orlando, FL, USA: IEEE Press, 1994.

[17] Sandhu, Ravi S., "The Typed Access Matrix Model." In *Proceedings of the 1992 IEEE Symposium on Security and Privacy.* Oakland, CA, USA: IEEE Press, 1992. p. 122-136.

[18] Sekar, R., Yong Cai and Mark Segal, "A Specification-Based Approach for Building Survivable Systems." In *Proceedings of the National Information Systems Security Conference*. October 1998.

[19] Simon, Rich and Mary Ellen Zurko, "Adage: An architecture for distributed authorization." Technical report, Open Group Research Institute, 1997. http://www.opengroup.org/www/adage/adage-arch-draft/adage-arch-draft.ps

[20] Woo, Thomas Y.C., Simon S. Lam, "Authorization in Distributed Systems: A Formal Approach." In *Proceedings of the 1992 IEEE Symposium on Security and Privacy.* Oakland, CA, USA: IEEE Press, 1992. p.33-50.