A Graph-based Language for Specifying Security Policies

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

Abstract

A security policy states the acceptable actions of an information system, as the actions bear on security. There is a pressing need for organizations to declare their security policies, even informal statements would be better than the current practice. But, formal policy statements are preferable to support (1) reasoning about policies, e.g., for consistency and completeness, (2) automated enforcement of the policy, and (3) other formal operations on policies, e.g., the composition of policies. We present LaSCO, the Language for Security Constraints on Objects, in which a policy and a system description are independently specified. LaSCO policies are specified as expressions in a quantifier-free logic, but also, to provide visualization, as directed graphs. Formal semantics have been defined. LaSCO can be used for policy requirements that conform to its very general object-based system model. The implementation of LaSCO is in Java, and automatically generates wrappers to check Java program executions with respect to a policy.

Keywords: computer security, security policies, access control, formal policy specification

1.0 Introduction

Security forms a core component of many systems. Traditionally, what is meant by security has been formally described for certain interpretations of confidentiality, integrity, but less precisely for availability. However, what is realistically meant by security for a particular system[2] varies from system to system, and possibly depending on the interaction of the system with its environment. The military would likely have a different definition of security than a bank, a university, or a

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

2. We use "system" generally here, to include any computational entity. In particular we can model a program execution, a file system, an operating system, a workgroup, and a network of hosts.

home user would. They each have their own needs that should be reflected when securing their systems. In addition, one can describe security at various system levels. A security policy is a description of the security goals for a system and how a system should behave in order to meet these goals. An example policy is that the request for a purchase and its approval must be from different users, each authorized for the particular operation.

Since the security goals for a system effect, among other things, how the security mechanisms on the system are configured, it is important for the security policy to be stated clearly and precisely. Additional benefits accrue from the precise formulation of a policy that can be directly used to configure the security mechanisms or can be used to formally reason about the effect of the policy. A common security need is to restrict access to the resources on a system. This is reflected in a constraint security policy[3]. These constraints describe, in general terms, restrictions on how the system should behave. We focus on constraint security policies in this paper.

It is inadequate to formulate policy just in terms of operating system objects such as files and network connections. For some situations, we need to state policy at a level beyond which an operating system has good knowledge, for example at the application level. To an operating system, an object in an application is not necessarily visible and the operating system would in any case have minimal knowledge of the semantics of an object - what it means and represents. Thus, it is important to state policy at the application level and lower levels.

Our view of policy is object-based. Typically a system would contain one or more objects that do not exist in isolation. They interact through events such as accesses and communication. Policy places constraints on the events that interrelate the objects and on the objects themselves. A particular policy is in effect over a *domain*. This domain determines when the policy is relevant. This might reflect the state of an object, that a particular event has occurred, or that some set of events has occurred. Once we have stated where the policy is relevant, the *requirement* of the policy becomes applicable. The requirement might place a restriction on the events such as requiring a certain type of event to be used or restricting certain events, it might place a restriction on the state of an object, or some combination of these.

In this paper, we define a language that we call LaSCO (the Language for Security Constraints on Objects) which represents policy as a directed graph with annotations. We believe this visual form of policy formulation is convenient, and show the relevant graph operations to policy manipulation. LaSCO can be used to state policies for many types of systems, including operating systems and programming languages, particularly if they are object oriented. The policy is specified separately from a program. The language is not tied to any particular enforcement mechanism and the policies so stated may be used in different ways. For example,

3. Other types of security policies are ones that describe how trust is managed, describe how to choose the security options in setting up a network connection, and ones that describe how to respond to a security violation.

policies may be compiled into a program, used by a reference monitor [1] or wrapper to mediate access to system objects, or to check an audit trail off-line.

We can state policies that apply whenever a certain pattern of accesses is encountered. This can be seen as a template for a general policy rather than configuration details that an access control matrix describes. The language can describe policies that span events over a period of time and can describe policies that depend on conditions other than the state at the time of check.

The language is visual which might make it easier for humans to cope with, a serious issue since policy specification will be human intensive. At the same time, our language has a formal basis. Having a formal basis and a formal semantics promotes our ability to reason about policies themselves, and about system with respect to a policy.

This paper is organized as follows: Section 2.0 presents an overview of our approach. In Section 3.0 we present our model of the system to which a LaSCO policy is separately applied, and in Section 4.0 we present the LaSCO language and its semantics. We present several examples of LaSCO policies in Section 5.0. Section 6.0 presents ways of using LaSCO for particular system such as programs and file systems. In Section 7.0 we discuss the expressiveness of the language and Section 8.0 presents a comparison of our work with related work. Section 9.0 concludes and discusses future work.

2.0 Overview

This section provides a brief overview of two aspects of our approach: the type of system that we can model, and the LaSCO policy language. We expand on this description in the subsequent two sections.

We describe security policies using a formal language based on directed graphs. The system to which the policy is applied consists of a series of events, each occurring between a pair of objects, some of which can be viewed as a subject. We apply the policy by identifying portions of a system to which the policy applies and checking the requirement part of the policy on that portion of the system.

2.1 System model overview

We first describe how systems are modeled. A system denotes an execution of the environment (system) upon which policy is applied. During this execution, computation events occur among active objects. We can thus model a variety of systems including programs, operating systems, file systems, and networks.

An object is a system entity that has state and that might invoke or be the target of an event. An event is an access or communication (signal) between a pair of objects at a particular time. A system is, thus, an ordered set of temporal instances. A system instance is a snapshot of the system at a moment in time; it contains a set of active objects and a set of pending events.

We present an example system in Figure 1 in which there are four objects and

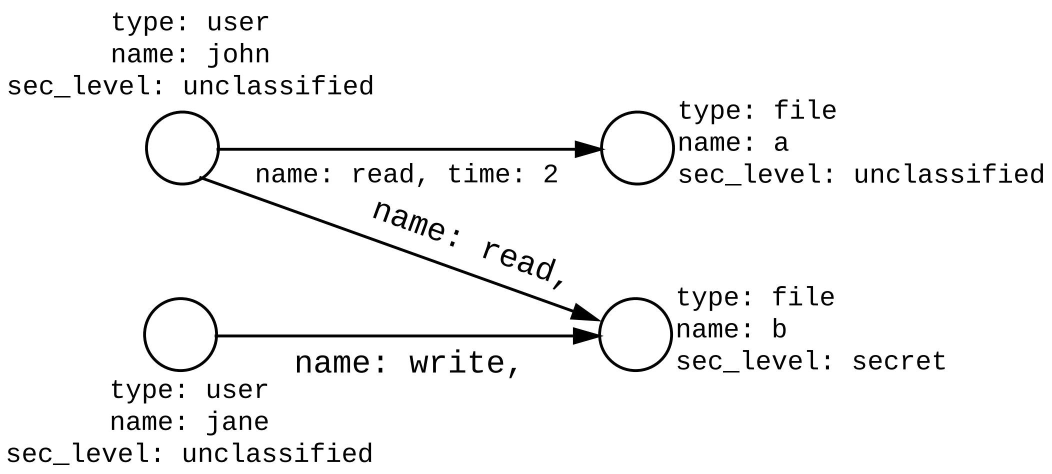


Figure 1. Example system depicted as a graph. The nodes represent objects and the edges events.

three events. Each object has a set of attribute-and-value pairs that represents its current 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, events have 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. This denotes three events that occur between four objects; whether these events are allowed is defined in the policy.

2.2 Policy language overview

A constraint policy language describes constraints on a system that must hold when the system is in specific states. This state may include events that have occurred on the system. 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*). A security policy is stated by asserting that if a part of the system is in a specific state, the events and objects in that part must satisfy a set of properties.

The simple (access control) security property of Bell-LaPadula [2] specifies that if a user is reading a file, the security level of the user must be at least as great at that of the file. We introduce our policy language by depicting this policy as a LaSCO policy graph in Figure 2.

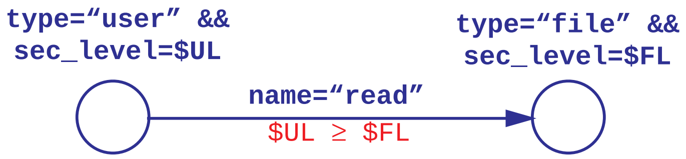


Figure 2. Policy graph for the simple security property

Policy graphs are annotated directed graphs. The **`bold text`** associated with (and depicted near) a node in the graph describes the particular kind of object the node represents. The formal annotation can be viewed as a boolean test for whether an object fits the policy, evaluating to true if the attribute values for the object permit it to be satisfied. For example, the right node in the policy graph represents a user with a particular clearance level, the test evaluating to true for an object if its *type* attribute has the value "user" and if its *sec_level* attribute has the same value as the variable UL (whose value is bound by this reference). The **`bold text`** associated with an edge describes the particular kind of event the edge represents and is a boolean test for whether an event fits the policy. The annotation evaluates to true if the parameter values for the event permit it to be satisfied. The edge in Figure 2 represents a *read* event and matches a event when the value of the name event parameter is "read." The `standard text` annotation is the requirement for the policy. Here it is a simple test of whether the variable UL is greater than or equal to the variable FL. Given the prior use of these variables, this is effectively a test of whether the user *sec_level* attribute value is at least as high as the file *sec_level* attribute value. If this is true, the requirement is satisfied and the policy upheld for that part of the system. Otherwise the policy has been violated.

Let us now consider this policy in the context of the example system depicted in Figure 1. We depict this linkage in Figure 3. When the domain of the policy is

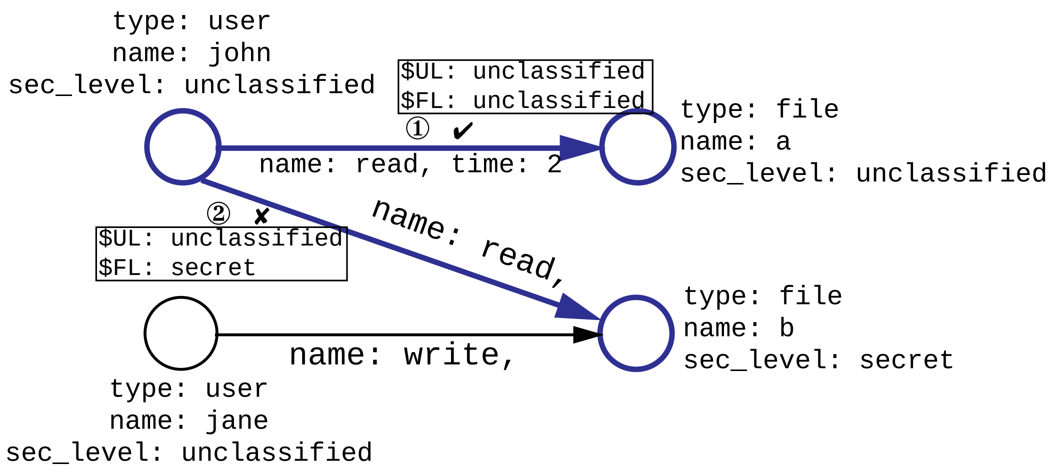


Figure 3. Depiction of simple security property applied to the example system. The two places where the domain applies is noted along with the necessary variable bindings.

applied to the system, the policy graph matches in two locations, indicated by the ① and the ② and the thick lines. The policy requirement is satisfied by ① but not ②. In the first case, the binding of the variables from the domain is that UL and FL both have the value "unclassified". This meets the requirement for the policy. However in the second case, UL has the value "unclassified" and FL has the value "secret", which causes the requirement test to evaluate to false, signalling the failure of the requirement to be satisfied.

3.0 The system basis for LaSCO

Policies are in effect over some system. We model the system upon which a policy is applied so that we have a construct upon which the definition of LaSCO is based. With a system model, we can define the meaning of LaSCO as it applies to an arbitrary system that fits the system model.

The LaSCO system model is object-based. It consists of a set of objects and a set of events that occur between the objects. Events have unchanging named parameters associated with them. Objects consist of a fixed set of named attributes which describe the security-relevant state of the object. The values of these attributes may vary over time. However there is an unique ‘id’ attribute associated with each object that does not change.

We assume that events and changes to object attribute values occur at discrete time steps and that multiple events can be happening simultaneously. This allows us to view our system 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 which are awaiting approval. It is at this point where policy is considered for a system. We denote the time of a system instance in each of its events using a *time* parameter.

4.0 Access control policy language

In LaSCO, policies are represented visually by a ***policy graph***. In this section, we discuss the syntax of LaSCO and its informal semantics. For a formal treatment of the language, including its formal semantics in first order logic, see [10].

4.1 Predicates

The text annotations we introduced in Section 2.2 are termed ***predicates***. There is a ***domain predicate*** and a ***requirement predicate*** for each node and edge in the policy graph. (Nodes and edges without an explicit domain or requirement predicate have a default “True” predicate.) Although they serve different roles, with domain predicates describing the object or event that is relevant to a part of the policy and requirement predicates describing what must hold if the domain is satisfied, they are evaluated in the same way. In either case predicates are patterns for the attributes of an object or the parameters of an event. For the moment, let us consider only predicates without variables, which we will term *simple predicates*.

A simple predicate is a boolean expression formed from attribute or parameter names and constants combined by operators from a certain set of various logical, comparison, set, and mathematical operators. Parentheses may be used to nest subexpression. 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.[4]

4.2 Domain and requirement

As represented in the policy graph, the domain of the policy (a pattern for where the policy applies) is the set of domain predicates and the nodes and edges. The requirement (restrictions to check then) is the set of requirement predicates. To use the policy, we find the locations in the system where the domain matches, then check the requirement for each of these cases.

4.3 Domain matching

Let us now consider the process of matching the domain of a policy to a part of the system. We address this for simple domain predicates.

The domain pattern is satisfied (it matches) when each node and edge in the policy is satisfied by part of a system. As demonstrated in Figure 3, a part of the system that matches the policy consists of an object for each node and an event for each edge. Each of these objects and events is part of a one-to-one map between nodes and events and objects and events. In the simple predicate case, this mapping constitutes what we term a ***policy to system match*** (***match*** for short).

The objects and events in a match are related in a particular way. The objects matching the nodes at the ends of each edge must be those that are at the same end of the event that matches the edge. Specifically, the source node of an edge must match the source object of the event the edge matches and the destination node of an edge must match the target of the matching edge for the event. As the value of an object's attributes might change between system instances, we require that an edge's incident nodes must match their system objects in the same instance that the edge matches an event. As a node might be incident to multiple edges, this means that nodes must match their object's attribute values in each instance that an incident edge matches an event. In the case of isolated nodes that have no incident edges, they can match an object in an instance.

As the domain may apply in several ways in the system or not at all, applying the domain to the system produces a set of matches.

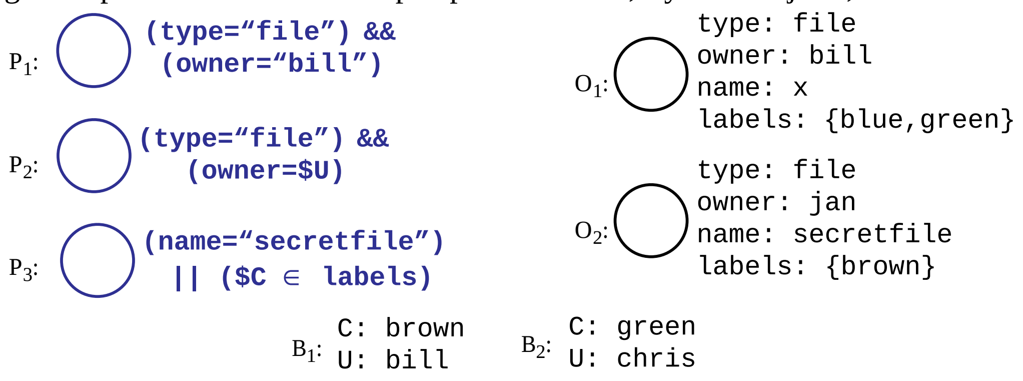
4.4 Variables

LaSCO policies make use of a set of a policy variables to relate different attribute values and parameter values of different objects and events. The scope of variables is a single LaSCO policy graph and each variable has a certain value in that scope, for a particular match. Variables may appear as operands in domain and requirement predicates and are denoted by a "$" prefix.

Variable bindings represent a set of policy variables that have values bound (assigned) to each of them. Predicates are evaluated in the context of a set of vari-

4. 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 to be satisfied by the object or event.

able bindings. This is in addition to the attributes or parameters mentioned in Section 4.1. To evaluate the variables in a predicate, we substitute the value of a variable in the bindings for its name. We demonstrate predicate evaluation here. Figure 4 presents several example pattern nodes, system objects, and variable



P1: (type="file") && (owner="bill")
P2: (type="file") && (owner=$U)
P3: (name="secretfile") || ($C ∈ labels)
O1: type: file owner: bill name: x labels: {blue,green}
O2: type: file owner: jan name: secretfile labels: {brown}
B1: C: brown U: bill
B2: C: green U: chris

Figure 4. Example pattern nodes, system objects, and variable bindings. P_1, P_2, and P_3 are pattern nodes, O_1 and O_2 are system objects, and B_1 and B_2 are variable bindings.

	O_1	O_2
P_1	satisfied with any variable bindings	not satisfiable by any variable bindings
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 5. Predicate evaluation example. The table depicts which variable bindings satisfy each pattern node's predicate

bindings. The table in Figure 5 uses these to give examples of evaluating a predicate on a domain pattern node in the context of a system object and particular variable bindings.

We mention now two restrictions on what may and may not be found in predicates. Each variable present in the policy must be in some subexpression of some domain predicate as <variable>=<value> (or <value>=<variable>) where <value> is a (possibly derived) single value. This subexpression may not be part of a disjunction. This ensures that all variables have a single value for the domain. The second restriction is that node requirement predicates may not contain any attribute references[5].

The domain is satisfied when all of its nodes and edges can simultaneously be satisfied by a set of variable bindings. When a policy is being applied to a system, it is the mechanism doing this application that determines the variable bindings. We can now observe that a policy to system match, in addition to containing map-

5. The values of an attribute might change during the 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 significant attributes only match the policy in one particular way while holding a particular value. This restriction could be narrowed to nodes that have more than one incident edges without ambiguity arising.

pings between nodes and edges and objects and events, also contains the set of variable bindings that enable the mapping.

4.5 Requirement checking

We check the requirement of a policy against a match by evaluating each requirement predicate. A node requirement predicate is evaluated with the attributes of the object that matched the node in the domain and the variable bindings from the match. Edge requirement predicates are likewise evaluated in the context of the parameters of the event that matched the edge and the same variable bindings. If each of the requirement predicates evaluates to true, the policy has been upheld, otherwise the policy has been violated. The result of applying a policy to the system will return how the system violates policy; the implementation entity doing this checking may use this knowledge as appropriate for its situation.

4.6 Policy composition and operations

We have defined some operations on policies in [10], which we summarize here. Policies can be composed through *conjunction*. Each constraint policy must be upheld for the composed policy to be upheld. If any of the policies are violated in the system, the set of policies has been violated. *Disjunction* between a pair of policies implies that in cases where both policies' domains apply, only one of the requirements need be met. Otherwise the policies may be considered separately.

The *nullification* of a policy takes it out of effect, forcing its requirement to always be satisfied. *Reversing the requirement* of a policy negates (reverses) the requirement of policy, while leaving the domain unchanged. In the general case, to express the result in terms of LaSCO policies requires a set of conjuncted policy graphs. A policy *contains* another policy if it is the case that the first policy enforces the constraints of the second policy in all situations.

5.0 Examples

This section presents several examples of using LaSCO to describe different kinds of 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 2. Access control lists (see [5]) and access control matrices [11] can be represented in LaSCO with one or more single-edge policies. For example, Figure 6 depicts the following policy: if subject "sam"

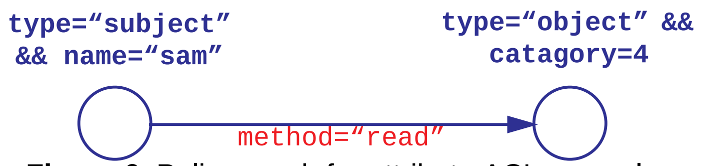


Figure 6. Policy graph for attribute ACL example

is accessing an object in *category* 4, then the access must be a read.

Role-based access control (RBAC) (Sandhu, *et.al.* [14]) can also be represented in LasCO, as it is similar in form to access matrix constraints, with the main difference being that the subject is a role and not a user. In modeling the 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 7 denotes the RBAC policy that only subjects that

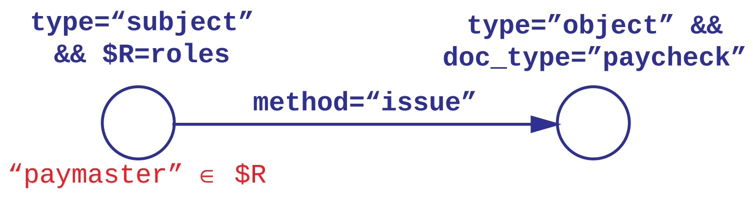


Figure 7. Policy graph for payroll RBAC example

have the role of paymaster, can issue a object of document type paycheck.

We can also state policies where we need to check a requirement when we have multiple events seen together in a certain pattern through LaSCO policies with multiple edges. Consider the Chinese Wall policy [3]. The idea behind the Chinese Wall policy is to prevent conflict of interest situations by consultants that may be employed by a number of parties with competing interests. The policy achieves this by forbidding someone from accessing data from different parties where the parties are in the same conflict on interest class. This is depicted in LaSCO by a node with two edges originating from it as shown in Figure 8. The middle node is

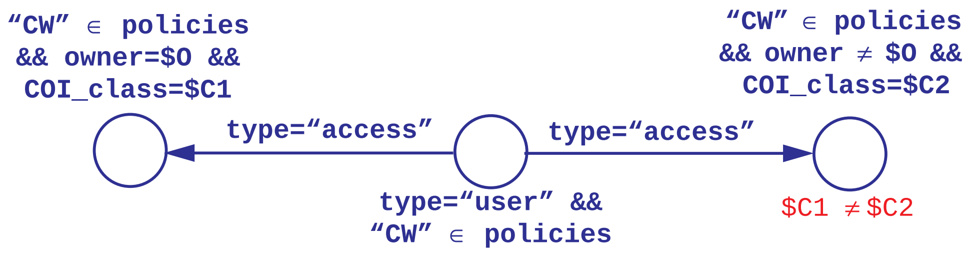


Figure 8. Policy graph for the Chinese Wall policy

a "consultant" whose accesses are limited by the Chinese wall policy. The edges from the consultant node represent accesses to sensitive objects with different owners that are subject to Chinese Wall. The constraint is that the owners of these objects cannot be in the same conflict of interest class, stored in the attribute "COI_class".

Separation of duty policies can be depicted in LaSCO. An example is shown in Figure 9. This depicts a policy for a system where there is a separate function for

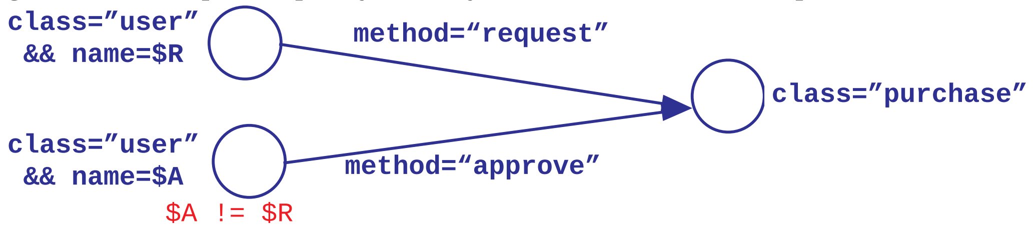


Figure 9. Policy graph for purchase request and approval separation of duty

requesting policies and getting 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 maximum number of accesses of a particular type that are allowed. As an example of this, consider the LaSCO policy depiction in Figure 10, where we show a policy that might be relevant to a program that

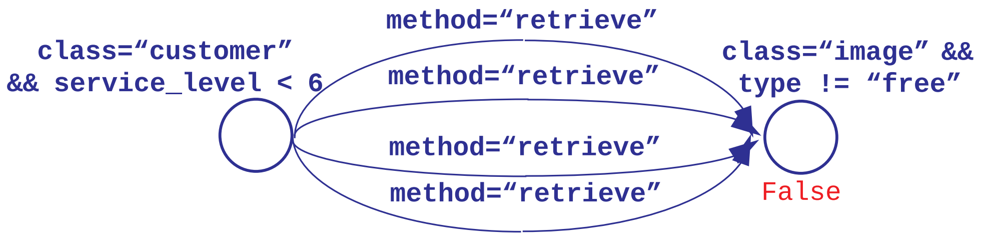


Figure 10. Policy graph for image retrieval quantity restriction

accesses images from a 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 these is no way to uphold the policy at that point. (As syntactic sugar, we have considered an extension to LaSCO that adds an iteration count to edges.)

LaSCO states restrictions just on objects in form of policy graphs that contain one or more isolated nodes. We present an example of this in Figure 10. The policy

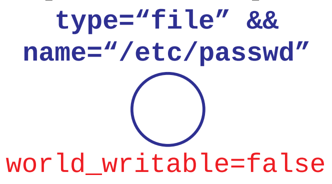


Figure 11. Policy graph for password file restriction

in this figure states that the “/etc/passwd” file should never be world writable.

6.0 Applying LaSCO to real systems

In order to put into effect a policy described in LaSCO, we need to translate the policy into enforcement mechanisms. This can be automated and we discuss applying LaSCO to programs and to file systems as examples of this.

6.1 Using LaSCO to enforce policy on a program

We may use LaSCO to describe policies to be enforced on the execution of an object-oriented program. In the system being represented, objects are instantiated and methods are invoked to perform computations. Access control, thus, involves establishing constraints on object instantiations and method invocations. For this type of system, program objects are the objects for the system model and method invocations are the events.

Note that most traditional programming languages do specify mechanisms (such as types and public/private methods and variables) for controlling accesses to object states and methods. However, this is not dynamic or dependent on the state of an application. Note that while it may be possible to add constraint checks directly in a program, for example, while writing the program, there is added benefit from stating the policies separately using a formal method. These include allowing some degree of independence between the program and the policy, facilitating better understanding of the effect and intention of a constraint through a directed language, and enabling reasoning about policies and interactions, independently of a particular system.

We now describe the manner in which access control policies might be derived, specified and enforced. In Figure 12, we show the steps that might be taken by a

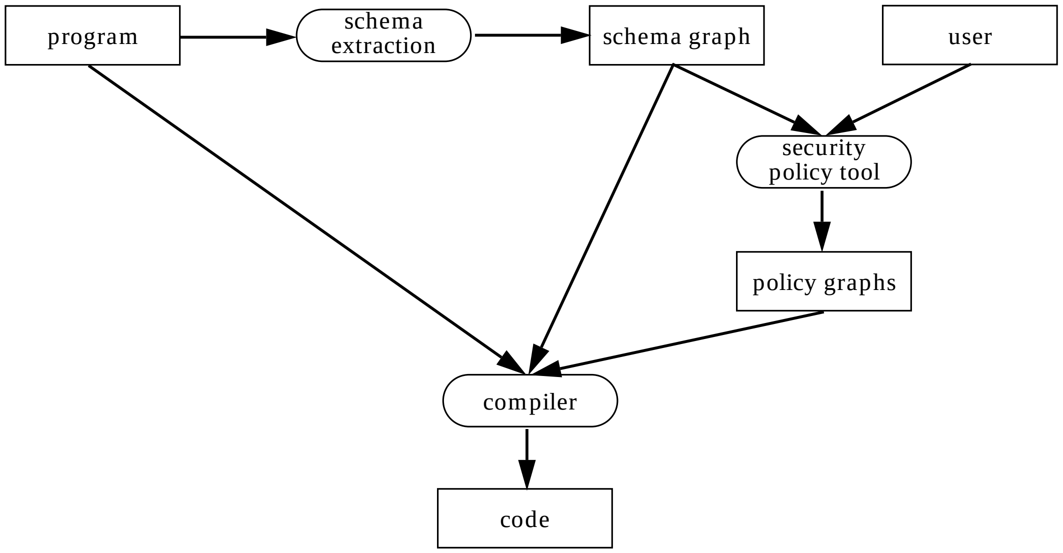


Figure 12. Access specification and control for policy application at the language level.

user to specify access control policies:

- Schema Extraction: Given an arbitrary program, a schema extraction tool constructs a program schema graph from the program. Nodes of the program schema graph denote class definitions of the program, whereas edges between the nodes represent method invocations or object instantiations found in the program source. Edges, thus, capture the various access relationships.
- Security policy tool: The user selects a portion of the program schema graph using a security policy tool. She then constructs a LaSCO policy graph by adding constraints over the schema graph. In addition, the security policy tool will provide a library of security policy graphs that the user can customize for the schema. This leads to the creation of a set of policy graphs.
- Compiler: 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.

We have an implementation of this approach underway for Java, making use of an existing Java parser. The graphical user interface is for X-windows. The enforcement of policies is achieved through wrapping portions of code with policy checks. Certain parts of the policy can be evaluated statically and are used to reduce the amount of run-time checking needed.

6.2 Using LaSCO for file systems

We can describe access control policies for a file system using LaSCO. To do this, we might model the file system as consisting of files and subjects (for example, users or processes), both of which would be objects in the LaSCO system model. Files objects would naturally have attributes such as its owner, type, size and modification date, and the subject objects would have the appropriate security relevant attributes for the system. The system events would include accesses by subjects to the files on the system with the details of the access as parameters. Policies for this system thus might restrict access to files based on the details of the access, what is being accessed, and what has previously occurred on the system. Policies also might restrict the state of files.

The policy engine checking the LaSCO policies might be situated in different ways with respect to the system. It could serve as a reference monitor, moderating requests to the file system from applications. It might also be along side the file system, being called when a policy decision is needed. A third possibility would have the access checking done by an application such as an intrusion detection system that scans over an audit log of accesses and object states, perhaps permitting the use of LaSCO policies to be retrofitted onto a system.

7.0 Discussion

Though aimed at specifying security policies for access control, LaSCO has a far more general model than traditional access control mechanisms. LaSCO can specify general system constraint policies, such as access control, limiting the actions of users, and system assertional checking, that impose a condition that may be based the current state of the system and events that have occurred previously in the system.

The system model employed by LaSCO is simple, very flexible, and we find it adequate for modeling security aspects of systems. It fits the traditional notion of security where subjects and objects interact through accesses and communication. Single-edge LaSCO policies permit the specification of simple access control policies where all the information needed to make the decision is available in the current system state and events. LaSCO policies that make reference to multiple events permit access decisions to be based on what has previously occurred. Policies can also impose constraints on the state of an object. As a result of the flexibility of LaSCO, an application developer or site can create custom policies to fit their needs. This ability promotes security and is in contrast to some policy mechanisms which only allows a limited number of policies to be enforced.

Some constraint policies cannot be stated in the current LaSCO in part because we wish to keep the language simple initially. This includes policies that impose requirements to the effect that certain events are required to occur. This is the case for the policy that employees must execute orders given by their supervisor and is also the case when we require a minimum number of events to occur. Another type of constraint policy that cannot be stated is when the policy refers to an object having to be in two or more distinct states through the system. We have not seen a natural instance of this sort of policy though. In LaSCO, it is not possible to express the fact that in order for the policy to be applicable certain events should have not occurred. An example of this is policies that refer to events that occur without other events having occurred. While LaSCO policies state what a policy violation is, they do not state the response to take when a policy violation occurs. This might be an important part of enforcing the policy. We have considered several extensions to LaSCO to address these limitations [10].

8.0 Comparisons with other work

The work of this paper most closely resembles the Miró work of Heydon, Tygar, Wing, *et. al.* at Carnegie Mellon University. Miró consists of two languages, an instance language and a constraint language [9], both of which are based on Harel's hierarchical graphs [8]. The instance language's formal semantics is defined in [12] and is analogous to our graphical depiction of the system that we formally define in [10]. It describes a file system access control matrix. The constraint language, which describes security constraints on the file system, contains a domain part (called an antecedent) and a requirement part (called a consequent) and has a predicate on nodes which is similar to LaSCO's predicates. Miró can only express allowable states (a snapshot of a dynamic system), whereas LaSCO is more flexible. Also, LaSCO can be applied to systems other than file systems and has simpler semantics.

Access matrices [11] and the related access control lists and capabilities (see [5]) are a traditional means of specifying security permissions. LaSCO can represent this type of security constraint, but overcomes some of its limitations. In stating a policy, LaSCO can denote a specific kind of objects and event by their (dynamic) attribute value, whereas the granularity of access matrices is specific subjects and objects. LaSCO can include historical context in its access control which access matrices cannot do due to their static nature. Other approaches based on access matrices aim to overcome some of its limitations. TAM [15] introduces safety properties into an access control matrix through the use of object typing and defined sets of operations to execute under different conditions. BEE [13] is an access control mechanism where decisions are based on the result of a boolean expression evaluation for an access right. When making decisions, BEE cannot make reference to other events that have occurred. A goal BEE shares with LaSCO is to allow users to think at a policy level when implementing restrictions. The Authorization Specification Language is a similar approach [7] that expresses the desired authorizations regarding user access to objects in logic and has conflict resolution rules defined in the language. The goal with this is to enforce varying security policies without changing the security server. This more structured approach towards overall site policies than LaSCO currently defines is useful in certain situations.

Cholvy and Cuppens [4] express the policies on a site in terms of deontic logic, which states what is obliged to occur, what is permitted to occur, and what is forbidden to occur and how to deal with inconsistencies. This seems to be a more general approach than in LaSCO (in which what is obliged is found in the requirement). However, the approach is limited to expressing policies for agents in terms of what they the obliged, permitted, and forbidden from doing. An approach towards specifying authorizations for subjects over objects that is thoroughly formal is presented in Woo and Lam [17]. Precise semantics, allowing for inconsistency and incompleteness in specification, and distributed specification for their logical language are present there. However for this, it is not as clear how to implement the policies, for example, for an application. Logic-based approaches, while strong in some ways such as the ability to reason about problems, are often weak in terms of the ability of the user to effectively interact with policies so stated due in part to the background in logic that is needed and the sometimes difficult intuition. With LaSCO, we address this limitation through a visual presentation.

The Adage architecture [16], developed at the OG 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 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 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.

Deeds, developed by Edjlali, Acharya, and Chaudhary [6], 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 plan to do, 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.

9.0 Conclusion and future work

In this paper, we have presented a formal policy language based on graphs. Separately specifying a system and a policy is possible in LaSCO, where, one can specify constraint policies which constrain accesses, communication, and object state on a system under given circumstances. The specific constraint might depend on the context in which an access occurs. Though designed for security policies for access control, LaSCO may also be useful for other environments such as software development where behavioral assertions might be checked at run time. The language can be used on any system that conforms to our system model (most of which do), and we presented means of using LaSCO to specify policies in an object-oriented program and a file system. The language and system model was described, its semantics presented, several examples shown, and we discussed what it could and could not specify.

LaSCO has the benefits of allowing policies to be stated separately from the system, of being enforcement mechanism independent, and allowing policies to be specified using flexible patterns that might include multiple events. It is a formal language with formal semantics and allows reasoning about policies in the language and the system they are enforced on. Its visual basis might make it easier for users to specify policies in LaSCO.

Though we have omitted the details from this paper due to space considerations, we have developed a formal semantics of LaSCO and operations on policies [10]. We are implementing LaSCO in Java programs through instrumenting the programs with policy statements checked at run time. In this environment, we can take advantage of restrictions on classes of objects and on the particular event that is involved for the policy. We will continue our implementation as future work. We will also study LaSCO further, especially in the area of policy composition and operations and policy conflicts.

10.0 References

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

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

[3] 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.

[4] 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.

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

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

[7] 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.

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

[9] 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.

[10] 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.

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

[12] Maimone, M.W., J.D. Tygar, and J.M. Wing, “Miró Semantics for Security.” In *Proceedings of the 1988 Workshop on Visual Languages*, Oct 1988. pp. 45-51.

[13] 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.

[14] 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.

[15] 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.

[16] 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

[17] 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.