

Developing Dynamic Security Policies*

Prasad Naldurg, Roy H. Campbell, and M. Dennis Mickunas
Department of Computer Science,
University of Illinois at Urbana Champaign
IL, 61801, USA
{naldurg,roy,mickunas}@cs.uiuc.edu

Abstract

In this paper we define and provide a general construction for a class of policies we call dynamic policies. In most existing systems, policies are implemented and enforced by changing the operational parameters of shared system objects. These policies do not account for the behavior of the entire system, and enforcing these policies can have unexpected interactive or concurrent behavior. We present a policy specification, implementation, and enforcement methodology based on formal models of interactive behavior and satisfiability of system properties. We show that changing the operational parameters of our policy implementation entities does not affect the behavioral guarantees specified by the properties. We demonstrate the construction of dynamic access control policies based on safety property specifications and describe an implementation of these policies in the Seraphim active network architecture. We present examples of reactive security systems that demonstrate the power and dynamism of our policy implementations. We also describe other types of dynamic policies for information flow and availability based on safety, liveness, fairness, and other properties. We believe that dynamic policies are important building blocks of reactive security solutions for active networks.

1. Introduction

Policy Management tools provide administrators the ability to specify, implement, and enforce rules to exercise greater control over the behavior of entities in their systems. In this article, we describe the policy development life-cycle for a special class of policies we call *dynamic policies*. Without loss of generality, we model shared resources or entities in a distributed system as objects that export well-defined interfaces. The behavior of an object is controlled

by its interface. These interfaces allow other objects (including objects acting on behalf of users or administrators) to query, access, and modify the objects' operational parameters (or state variables) by calling the appropriate methods, thereby changing the behavior of the system.

Policy management tools automate the task of changing these parameters by providing a simplified cross-platform front-end to system implementations. Currently, most network policies are implemented by systems administrators using tools based on scripting applications [5, 26] that iterate through lists of low-level interfaces and change values of entity-specific system variables. The policy management software maintains an exhaustive database of corresponding device and resource interfaces. With the proliferation of heterogeneous device-specific and vendor-specific interfaces, these tools may need to be updated frequently to accommodate for new hardware or software, and the system typically becomes difficult to manage. As a result, general purpose low-level management tools are limited in their functionality, and are forced to implement only generic or coarse-grained policies [33].

Since most policy management tools deal with these low-level interfaces, administrators may not have a clear picture of the ramifications of their policy management actions. Dependencies among objects can lead to unexpected side effects and undesirable behavior [24]. In Seraphim [23, 7], we remedy this situation by focusing our attention on a special class of policies that are designed with explicit knowledge of system behavior and interactions between various system objects. Our policy development cycle begins with the formal specification of system properties of interest. These properties correspond to security guarantees we want to preserve in our system. Properties are represented as sets of desirable behaviors, described in terms of objects and methods, and are expressed using an appropriate formal notation.

Next, we specify a model of the system behavior with respect to the properties we want to guarantee. This model can be viewed as an abstraction of behavior of the system

*This research is supported by DARPA BAA 98-03 and AFRL Contract Number F30602-98-1-0192

implementation. This specification includes the behavior of all objects that correspond to identifiers in the property specifications, and the transitive closure of the objects that interact with them. We take advantage of model checking techniques [9] to verify that our system specification can satisfy the desirable behavior specified by the properties. Once this is verified, these properties correspond to behaviors that can be guaranteed within the framework of our model. If the property cannot be validated, model checking provides counter-examples that can be used to improve the system design and implementation.

The mapping between validated properties and policy-preserving security policies follows from the specifications. Once the specifications are validated against the system model, we identify specific objects, variables, and methods in the system implementation corresponding to the state variables and mechanisms in the property specifications. These objects and methods automatically form a part of the property-preserving policy implementation and enforcement mechanisms. For example, we allow access to system resource if and only if our system has a rule in its access control rule database that allows the action. This property has to be guaranteed by any access control policy implementation at all times. To change an access control policy, we need to change the corresponding entry in the access control rule database. In addition, we need to guarantee that access is only allowed to objects that can provide authorization proofs. These objects, rules and mechanisms therefore automatically form a part of our policy management infrastructure.

Based on the discussion above, we introduce the concept of a *dynamic* or executable property-preserving policy. A dynamic policy is a program consisting of a set of guards and actions, created by our policy administrator. It encodes not only the logic to modify the system implementation to change operational parameters, but also includes all the necessary guards to enforce good behavior and prevent its misuse. For example, in the access control policy example, the guard can include proofs of authorization, and the commands are programs to change the access control rules. In our Seraphim active network prototype, these programs map directly to active capsules, and can be viewed as in-line policies [27]. These policies are managed, updated and changed by executing the appropriate capsules in a suitable protected software context (NodeOS or EE) [6]. We describe examples of guards and commands for different types of policies in this paper. Active capabilities are special guarded commands for access control policies. These guards and commands allow us to change operational parameters in the policy implementation, without causing undesirable behavior. Policies implemented in this fashion can make strong and verifiable guarantees about system behavior.

We believe that the real application of such policies is in the design of reactive security systems. By including formal analysis, verification, and validation in our policy development life-cycle, we reason about the effectiveness of our policies, and can therefore change operational parameters of dynamic systems with greater confidence. Situational policies in response to attacks can be developed as the system evolves in response to threats and exposures. Once the framework for a new policy type is in place, new policies can be created on the fly, following the specification guidelines. These policies can also be enforced instantaneously by sending and executing guard capsules over networks, during an attack window, to successfully mitigate the impact of an attack. In the course of this article, we describe a framework for specification, enforcement and implementation of these dynamic policies within the active networking context. However, our techniques are general enough to augment any distributed system.

In Section 2 of this paper, we define the class of dynamic policies and present a general method for constructing these policies. We also discuss the threat model for this class of policies. In Section 3 we give a detailed construction of dynamic access control policies that are annotated with authorization proofs, based on preservation of safety properties. In section 4, we provide a brief description of the Seraphim architecture and the implementation of dynamic access control policies in the context of active networks. We also explain why active networks can be used as a framework to develop, disseminate, and enforce such policies, and how these policies enhance the usability of the active networking paradigm. We also briefly describe an example application developed by the Seraphim group that allows an administrator to change between two different access control policy strategies by creating policies on the fly. Section 5 gives examples of other dynamic policies for information flow and availability in terms of safety, liveness, fairness and other properties. We summarize related work in Section 6 and conclude this paper in Section 7.

2. Dynamic Policies

In this section we describe a general method for design, specification, enforcement and implementation for dynamic property-preserving policies. We also describe the threat model and discuss the attack resilience properties of this class of policies.

2.1. Policy Development Life-Cycle

We describe the construction of dynamic policies in a systematic manner. An example of this procedure is the design of Seraphim's dynamic access control policies that

is presented Section 3. Our policy development life-cycle can be broken down into the steps shown in Figure. 1.

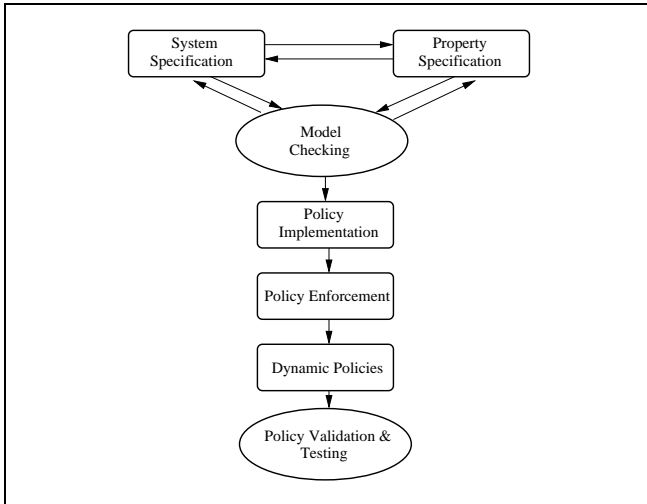


Figure 1. Policy Development Life-cycle

We explain each of these steps in detail below:

1. **Property Specification:** In the first step of our life-cycle, we specify the set of desirable behaviors or security properties that need to be guaranteed in our system, using a suitable language or formal notation. Traditionally, security policies are classified as *access control*, *information flow*, and *availability* policies depending on the type of behavior they control. Different types of properties include safety properties which assert that something bad never happens, liveness properties that assert that something (good) eventually happens, fairness properties that assert that everybody gets a chance to use a resource, etc. As described in the next section, access control policies can be specified as safety properties, and can be specified using simple temporal logic operators (LTL [9], PTL [32] etc.). We investigate the use of different types of logics and logic operators to represent properties for access control, information flow, and availability policies.
2. **System Specification:** This step models the environment of the entities of interest in the property specifications. Lamport [18] states that the behavior of every discrete system can be formally represented by a behavior. Dynamic systems can be modeled in different logics such as Rewriting Logic [2, 10] and TLA [17], by abstracting the behavior of objects and interfaces and mapping these as state variables and actions (e.g., as atomic propositions or predicates) in a formal notation. We develop a behavioral model of the environment of the property that includes not only all the

entities in the property specifications, but also any entities they interact with in the system. We also specify this model using an appropriate formal notation. For example, rewriting logic allows us to develop an executable specification that can be used directly for property checking. The level of abstraction of the model depends on the properties we are interested in, and Steps 1 and 2 are not strictly sequential. Modeling the behavior of the system helps us understand the different entities (or objects), their interfaces and interactions.

3. **Model Checking:** The next step is to check and verify that the properties can be satisfied by the behavior description of the model using model checking [9]. A Model M satisfies a property, expressed as a temporal logic formula f , if $M, s \models f$, where s is an initial state. This step helps us determine what security properties can be enforced in the system, and therefore allows us to identify *what dynamic policies can be implemented*, without sacrificing these guarantees. It is also a useful method to identify fundamental design flaws, to account for side effects and recognize subtle interactions that might induce undesirable behavior.
4. **Mapping and Identification of Policy Implementation Mechanisms:** In this step, we develop the mapping between logical variables in the temporal formulas of the property specification, and the system objects and interfaces of the system implementation. Policies can therefore be implemented by ensuring that the behavior of these objects do not violate the property specifications.
5. **Development of Policy Enforcement Mechanisms:** The policy implementation mechanisms identified in the previous step are augmented with *enforcers* to guarantee that they do not violate their behavior profiles. For example, enforcers for access control mechanisms can include reference monitors that intercept access requests and validate authorization proofs. An enforcer for a fairness policy can include a priority inversion mechanism that can be activated by an authorized user or administrator, to stop a misbehaving object from hogging resources. In general, the system may need to be augmented with enforcers to guarantee desirable behavior, and these form a part of the TCB (Trusted Computing Base). The system is engineered so that it is difficult to gain access or ownership to these enforcers or their interfaces.
6. **Creation of Dynamic Policies:** After the enforcers are in place, the next step is to specify the dynamic policies that can be implemented in the framework of our model and property specifications. These policies

are implemented as code capsules that encode the necessary guards and logic to change the operational parameters of the system objects, without sacrificing the properties.

- 7. Policy Validation and Testing:** The final phase in the policy development of dynamic policies is the testing of the dynamic policy implementation. So far, the formal specification and verification of properties, system models and dynamic policies allows us to reason formally about security guarantees. The testing phase actually checks the implementations of these policies to make sure they match the specifications. Software testing may involve type-checking and information flow analysis. Rigorous tests are developed from the formal descriptions of the model and properties and validated by observing the traces.

To summarize this subsection, we present a general construction for a class of security policies called dynamic policies that are based on formal specification, analysis, and verification of the behavior of systems. Dynamic policies provide administrators the ability to change the operational parameters of important system entities, without sacrificing guarantees of good behavior. These policies are implemented as programs and can be created on the fly and implemented by executing them in a suitable context. Our lifecycle helps us identify vulnerabilities and reduce the threat of exposure with respect to policy specification, implementation and enforcement. A concrete example of dynamic policies is given in the next section.

In the next subsection we briefly explain the threat model associated with our policy class.

2.2. Threat Model

Dynamic policies are designed with careful regard to system behavior. However, designing a system resilient to all threats and exposures will require similar guarantees from all components of the system, hardware and software, and adequate social engineering practices. Our policies provide a mechanism to change operational parameters to implement situational policies during the running of a system. The system does not need to be restarted and bootstrapped to change a policy rule. What we provide is a guarantee that changing the policy rules does not leave the system in an undesirable state. With our policies and enforcers, we augment systems with the mechanisms to ensure that the desirable properties that can be satisfied by the underlying system are enforced and not violated.

However, we are also limited by the level of abstraction in our model. While fine-grained abstractions can increase the resilience of our system, the performance penalties may

be unacceptable. Threats to our system can include external factors (e.g., social engineering, insider attacks, compromise of authorization credentials, malicious administrators, etc.) that cannot be modeled as undesirable behavior, because we may not be able to distinguish it from the desirable case.

3. Dynamic Access Control Policies

In this section we describe the specification, implementation and enforcement of dynamic access control policies. In the first subsection, we develop an initial specification of these policies following the general outline described in Section 2. We augment this with authorizations and present the complete specification in Section 3.3. In Section 4, we describe an implementation of these policies in the Seraphim architecture, along with example applications that demonstrate the attack resilience properties of these policies.

3.1. Access Control

In this subsection, we develop a behavioral model of the access control problem in a distributed system. Access-rights to resources are traditionally modeled as access control lists (ACLs) or capability lists. The basic construct used to represent access control is the access-right tuple $\langle \text{subject}, \text{object}, \text{right} \rangle$ which represents a subject's access rights to object interface methods. We restrict our initial specification to this type of access right, though our methodology is general enough to develop specifications for negative access rights, group and role-based access rights etc. ACLs are object-centric and list all the subjects that can access a given object, along with specific access rights. Capability lists are subject-centric and contain a list of objects and rights that can be accessed by a given subject. In most systems these two lists are merged into an Access Control Matrix, which is a table whose rows are subjects and columns are objects. At the intersection of each $\langle \text{subject}, \text{object} \rangle$ pair is the list of allowed methods that can be accessed by the subject. ACLs and capability lists are equivalent with respect to the types of access rights they model [8]. ACLs allow easy revocation and capability lists are easier for delegation, and the use of one or the other is a matter of preference.

An access control operation is therefore a simple lookup operation on the access matrix to check if the subject (or the object acting on behalf of the subject) is allowed to call the method on the relevant object. An access is allowed if and only if the corresponding entry can be found in the access control matrix. A formal specification of the basic access control rule is given as:

$$\boxed{Allow(s, o, m) \Leftrightarrow \langle s, o, m \rangle \in A}$$

Here A is the access control matrix and s, o, m stand for subject, object and method respectively. The desirable behavior of this system or the safety property that is invariant at all times is:

$$\boxed{\square(Allow(s, o, m) \wedge \langle s, o, m \rangle \in A \longrightarrow skip)}$$

Here \square is the *henceforth* or G operator in a suitable temporal logic (LTL or PTL).

From the modeling of the behavior of the system we observe that the safe behavior of the system relies on the proper enforcement of the property defined above. Access Control policy development consists of defining methods to control and modify the Access Control matrix A . Since the *Allow* method has three parameters (subjects, objects and methods), methods that manipulate these parameters form a part of the policy specification. Different access control policies can be enforced by adding or removing subjects, objects and methods. We use the guarded command language [12, 30] to specify set of executable policy management operations (or dynamic policies) in the system. A guarded command is represented as a (*guard* \rightarrow *command*) where a sequence of guards is followed by a sequence of actions. The guards specify the preconditions necessary to execute the commands. The system specification, with methods to add and remove users, objects and methods to A is given below (we call subjects users here) :

state vars

U: set of USERS initial ϕ

O: set of OBJECTS

A: set of $\langle u : USERS; o : OBJECTS; m : METHODS \rangle$

transitions

$Allowed(u, o, m) \wedge \langle u, o, m \rangle \in A \longrightarrow skip$

$AddUser(u, u') \longrightarrow U := U \cup \{u'\}$

$RemoveUser(u, u') \longrightarrow$
 $U := U - \{u'\};$
 $A := A - \{\langle u', o, m \rangle \mid o \in O, m \in METHODS\}$

$AddObject(u, o) \longrightarrow$
 $O := O \cup \{o\};$

$RemoveObject(u, o) \longrightarrow$
 $O := O - o;$
 $A := A - \{\langle u, o, m \rangle \mid u \in U, m \in METHODS\}$

$AddObjectMethodToUser(u, u', o, m) \longrightarrow$
 $A := A \cup \{\langle u', o, m \rangle\}$

$RemoveObjectMethodFromUser(u, u', o', m') \longrightarrow$
 $A := A - \{\langle u', o', m' \rangle\}$

From this specification we observe from the policy management interfaces that that any user in the system is allowed to add or delete subjects, methods, and objects arbitrarily. Furthermore objects and users can be added to the system, without creating any entries to the Access Control Matrix. However removing the objects implies that we need to delete all the corresponding methods. This policy implementation is of little use in the absence of stronger enforcement mechanisms. What is missing in this specification is the notion of authorizations.

In practical access control systems, different sets of users are allowed (or authorized) to create and delete users, objects and methods. In a DAC (Discretionary Access Control) system, only the administrators can create and delete new users. However, users are allowed to create and own objects and add access rights to objects they own. For example, if *user1* owns $file_{user1}$, then *user1* can insert $\langle user2, file_{user1}, read \rangle$ into A . In an MLS system, users and objects can be added only by administrators.

We augment our specification, to make it more meaningful, with special proofs of authorization. In order to change an entry in A , the user is required to produce a proof attesting that he or she is allowed, by some trusted authority, to actually call the relevant method. In order to verify this proof we need to add adequate policy enforcement mechanisms to our design. Since our aim is to prevent unauthorized modification of the capability lists, we add additional guards to our specification to provide the necessary mechanism to guarantee that only authorized modifications are allowed. These guard and action pairs that specify how to change capability lists are called “active capabilities” in Seraphim [7].

3.2. Trust Management

One way of generating such proofs is by using an attestation from a trusted administrator that gives the holder of the attestation the capability to change an access-matrix entry, or the permission to call a method to change the entry. This type of capability (also called a license [34]) or credential is an attestation of trust. These capabilities can be passed around among users and processes that are not running in the Trusted Computing Base. For this reason, they should be protected against modification. An attestation can have the same format as an access matrix entry, and can be made unforgeable by the issuer by attaching a cryptographic digital signature. The signature should tie in the name of the issuer and the intended recipient to prevent modification. This not only prevents modification, but also provides non-repudiation of ownership.

However, using these signed capabilities as attestations to control modification of capability lists is not without problems. Consider the set U of users who can issue signed capabilities, the set O of shared objects in the system and the set M of methods corresponding to access rights. The set of all licenses that can be presented to the policy manager in this system is exponential in the size of these three sets and is given by $C \subseteq U \times \mathcal{P}(O, M)$. In the absence of rules to govern the creation and dissemination of these licenses, the system can quickly become unmanageable. A user can have many different licenses and may present any subset of these to the policy manager during an access control request. The policy manager needs to decide whether the decision is consistent with the trust management implications of these attestations and this may be non-trivial. For example, the monotonicity of the privileges available after revocation may have to be maintained [34] to prevent undesirable behavior.

In our policy management architecture, we do not use signed capabilities to create the authorization proofs. Instead we rely on two simple credentials that attest to the identity of the entities (primarily subjects) and the ownership of one entity by another. The credentials in our system cannot be delegated and are not available to any entity except the policy implementation logic. An example identity credential $typeof(Alice, administrator)$ asserts that the identifier *Alice* is an administrator. The credential $owns(object, method)$ or $owns(user, object)$ attests that the method is “owned” or exported by the object or the object is owned by the user, respectively. Unlike signed capabilities, the size of these credentials is linear in the size of the number of users, objects and methods. All commands that modify the capability lists in the policy logic can include these credentials.

In the next subsection, we augment our specification with these credentials for a specific policy type (DAC) and show how these authorization proofs guarantee only authorized behavior in our system.

3.3. Modified Seraphim DAC Policy

Central to a DAC policy is the notion of ownership. Users are allowed to own objects and set appropriate access control policies for these objects. We associate this notion of ownership with a method *control* that gives the users the ability to delegate rights to access its object methods to other users. If a user has the capability $\langle user, object, control \rangle$, then it can add methods for the object in other capability lists.

In addition to this type of capability, we also have credentials or attestations of trust. From the policy enforcement point of view, we observe that access is allowed only when the corresponding capability can be found in the ca-

pability list A , and when the subject of the access control produces an authorization proof. We present the complete specification of Seraphim’s DAC policy next. The set C contains the identity and ownership credentials attested by the administrator, as required.

state vars

U : set of USERS initial ϕ

O : set of OBJECTS

A : set of $\langle u : USERS; o : OBJECTS; m : METHODS \rangle$

C : set of identity and ownership credentials

transitions

$Allowed(u, o, m) \wedge \langle u, o, m \rangle \in A \rightarrow \text{skip}$

$AddUser(u, u') \wedge typeof(u, admin) \in C \rightarrow$

$U := U \cup \{u'\};$

$A := A \cup \{\langle u, u', control \rangle\}$

$RemoveUser(u, u') \wedge \langle u, u', control \rangle \in A \rightarrow$

$U := U - \{u'\};$

$A := A - \{\langle u, u', control \rangle\};$

$A := A - \{\langle u', o, m \rangle \mid o \in O \wedge m \in METHODS\}$

$AddObject(u, o) \wedge owns(u, o) \in C \rightarrow$

$O := O \cup \{o\};$

$A := A \cup \{\langle u, o, control \rangle\}$

$RemoveObject(u, o) \wedge \langle u, o, control \rangle \in A \rightarrow$

$O := O - o;$

$A := A - \{\langle u, o, m \rangle\} \mid u \in U \wedge m \in METHODS\}$

$AddObjectMethodToUser(u, u', o, m) \wedge owns(u, o) \in C$

$\wedge owns(o, m) \in C \rightarrow$

$A := A \cup \{\langle u', o, m \rangle\}$

$RemoveObjectMethodFromUser(u, u', o', m')$

$\wedge \langle u', o', m' \rangle \in A \wedge owns(u, o') \in C \rightarrow$

$A := A - \{\langle u', o', m' \rangle\}$

From the specification, we observe that only administrators are authorized to add users to the system. The identity credential ($typeof(u, admin)$) is sufficient to guarantee this. In *RemoveUsers* we observe that the ability to remove a user depends on the ability to add a user, and by transitive closure, we can assert that only administrators can remove users. Any user can also add a capability for another user, as long as it owns the object and the object exports the required method. We observe from the transition functions *AddObject* and *RemoveObject* and *AddObjectMethodToUser* and *RemoveObjectMethodFromUser*, by transitive closure, that this ownership requirement is indeed enforced by the credentials ($owns(u, o)$) and ($owns(o, m)$).

To validate the model, for the given DAC specification,

we can reiterate that each transition that can change a capability list, along with the credentials and capabilities that are already in the system, under transitive closure, constitutes an authorization proof of why access should be allowed in our system. This is in accordance to the policy requirements. Any access control system built according to our specification is always in a safe state with respect to the manipulation of capability lists.

Other types of access control policies can also be specified and validated using a similar procedure. Seraphim's MLS (Multi-Level Security, which is a type of Mandatory Access Control) policy is based on the Domain and Type Enforcement (DTE) [1] policy. Users belong to domains and objects are classified into types. Policies are implemented using an access control matrix, where the access rights on object methods are stored at the intersection of a domain and type entry. We model this as a capability list indexed by the domain label, instead of the user identifier. The policy enforcer for MLS is responsible for preprocessing the $\langle subject, object, method \rangle$ tuple and converting it into a $\langle domain, type, object, method \rangle$ tuple required for DTE. Methods to add and remove users, domains, types, objects to types, and domains to users are restricted to the administrator by including a guard that verifies a $typeof(user, admin)$ credential. Adding types, objects, and methods to a domain require an additional $owns(object, method)$ credential. A partial specification of the DTE policy is given below:

state vars

U: **set of** USERS
O: **set of** OBJECTS
D: **list of** DOMAINS $\{d \mid d \subseteq \mathcal{P}(USERS)\}$ **init default**
T: **list of** TYPES $\{t \mid t \subseteq \mathcal{P}(o : OBJECTS)\}$
A: **set of** $\langle d : DOMAINS; (t : TYPES, o : OBJECTS); - m : METHODS \rangle$
DDT: **set of**
 $\langle d : DOMAINS; d : DOMAINS; m : METHODS \rangle$
L: **set of** credentials

transitions

$Allowed(d, t, o, m) \wedge \langle d, (t, o), m \rangle \in A \longrightarrow \mathbf{skip}$

...

$AddTypeMethToDom(d, d', t', m') \wedge typeof(d, admin) \in C$
 $\wedge owns(o', m') \in C \longrightarrow$
 $A := A \cup \{\langle d', (t', o'), m' \rangle\}$

$RemoveTypeMethFDom(d, d', t', m') \wedge \langle d', (t', o'), m' \rangle \in A$
 $\wedge typeof(d, admin) \in C \longrightarrow$
 $A := A - \{\langle d', (t', o'), m' \rangle\}$

In Seraphim MLS, we tightly couple the types and

objects, to maintain fine-grained control over the addition and removal of types and methods to the access matrix. Note that unlike DAC we do not require the $(owns(user, object))$ credential here. For a complete specification of Seraphim's dynamic access control policies, including RBAC, please refer to [29].

3.4. Model Validation

From the specifications for Seraphim DAC and MLS, we claim that if the credentials are generated correctly and the administrators keys are not compromised, then the executable policies allowed in our system have the required authorization proofs necessary (by transitive closure) to guarantee that authorized access to a resource is synonymous to the possession of unforgeable credentials.

In the next section, we describe briefly the Seraphim active network implementation of dynamic access control policies.

4. Active Networks and Dynamic Policies

In this section we provide a brief overview of our Seraphim security architecture for active networks. For more details refer to [23, 7, 22, 19, 20]. The major components of our architecture and their interactions in the context of the active network architecture are shown in Figure. 2.

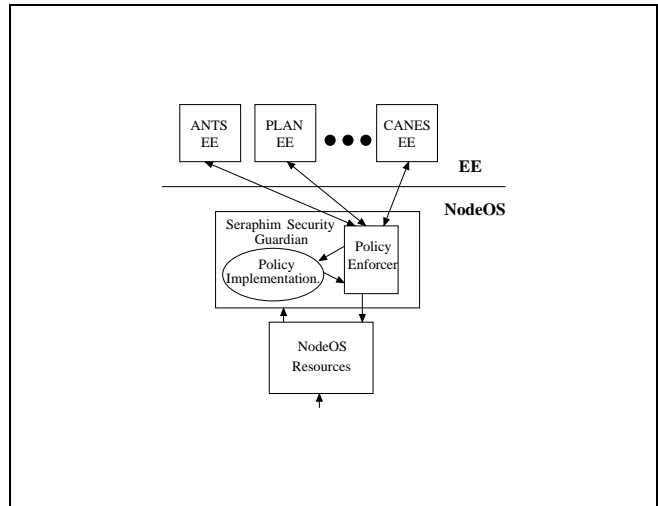


Figure 2. Seraphim Active Network Node

The key component of Seraphim Dynamic Policy Architecture is the Security Guardian. The guardian is the policy enforcement mechanism and is implemented as a colocated extension to the Node OS. Every node has a guardian that intercepts all accesses to node resources. The policy framework implementation is also a part of the guardian. The

implementation is componentized and reconfigurable, and can be downloaded dynamically when required.

To change operational parameters that change the policy implementation, administrators use the interface provided by the policy framework to create a customized piece of code that encodes the type of access control policy and the guarded commands from the policy specifications from the previous section. This code fragment is called the *active capability* (AC) [7, 23]. Unlike a traditional capability, which is merely a static authorization credential that encodes the principal and the permissions associated with the principal, an active capability is actually executable Java bytecode in our implementation¹. In addition, an active capability is protected by cryptographic digital signatures, resides in user space, and can be freely passed around.

An active capability relies on a policy framework for context. An application presents an active capability along with its regular data or protocol capsules to the active router’s guardian at execution time. The enforcement mechanisms in the guardian recreates the context of the policy type within its policy framework. If at any point during this process, the policy framework discovers that it does not have an implementation for the type of the policy, it downloads the code dynamically into the framework, using the underlying active network. It then instantiates the runtime parameters associated with the active capability in its sandbox-like environment and executes the active capability in this environment. Based on the result of the evaluation of this active capability, the access control decision is enforced.

From this description, we observe that active networks not only help in realizing the concept of dynamic policies, but also allow the development of “what you need is what you get (WYNIWYG)” implementations. In addition, the paradigm is enriched by these policies because their behavior can be validated by formal methods. One of the main concerns about active networking is the proliferation of code capsules that can cause arbitrary and undesirable behavior. Dynamic policies are examples of capsules that preserve a verifiable and well defined behavior. By deploying dynamic access control policies on chosen network routers, administrators can have greater control over what active capsules are allowed to execute and enforce stricter access control policies on these routers under attack.

In the next subsection, we describe an example application scenario which demonstrates the expressiveness and usefulness of these policies.

¹Note that this definition of active capability is less general than the one described previously

4.1. Example Dynamic Security Scenarios

Our dynamic policy architecture allows many different access control strategies to exist in the same system, though only one type of strategy may be active at any time. Within each strategy, the policy implementation can be changed dynamically without affecting the safety properties, while protecting against unauthorized modifications of these policies at the same time.

We have built two applications of example scenarios to demonstrate our dynamic access control policies from the specifications. The first example is the dynamic firewall.² Initially all routers are bootstrapped with the same access control policy. Then an attack scenario is simulated, where we generate an unwanted ICMP ping capsule directed at a victim router, forcing the victim to reply with another ICMP capsule. An attack detection agent triggers an alarm that dynamically removes the ability of a ping capsule to access the routing table implementation on the victim router. The victim router drops these packets without replying, and propagates a “vaccine” to its upstream router, which deletes its ICMP ping access rule and so on. Eventually, the attacker’s flood is stemmed at the source. Once the attack stops at a node, the access rule is reinstated. The dynamic firewall grows outwards from the victim, filtering packets closer and closer towards the attacker and remains in place only as long as the duration of the attack. The average overhead [7] for an application running on a Sparc 10 on the same 100Mbps average time to send and install a vaccine across the network is *34ms*.

Our second example demonstrates how we change the DAC policy on a specific computer to a more restrictive MLS policy to protect the integrity of information flowing between sensitive objects on the system. This policy can be deployed in response to an email virus. In the new MLS policy, users are prevented from sending messages on the network by removing their ability to transfer to the network domain and send their messages. This example is straightforward and requires that the policy logic of both MAC and MLS already exist on the system. Policies are developed and installed by the administrator on the fly, and when the administrator is done implementing the policy in the new strategy, the appropriate enforcement mechanism is activated. The performance overheads to switch between two strategies for our simple example was $\approx 2s$. Other examples of dynamic policies in Seraphim and detailed performance numbers can be found in [21, 20].

5. Information Flow and Availability Policies

Security policies are classified as access control, information flow, and availability policies [30]. As we saw in

²This application was demonstrated in [7]

Section 3, access control policies can be specified as safety properties (also shown in [30]). In this section, we examine the other two classes of policies and explore the type of properties and logic needed to specify these policies. Property types include safety, liveness, fairness and denial of service. We also include example specifications of these properties and briefly describe some enforcers for such policies. Since this section describes work in progress, we do not give complete specifications of systems.

5.1. Information Flow Policies

Information flow policies specify controls over flow of information among different classes. Information flow policies can be specified as either safety or liveness properties [30, 32]. An example of a safety information flow policy is the MAC policy that regulates the directional flow (send or receive) of certain classes of information. In this sense, a well-behaving information flow policy is a safety property where no bad flows occur. To enforce this policy, flow filters are added to the points where information flows in and out of the system objects (or ports). A partial specification of the Bell-LaPadula [3] information flow policy as a safety property is shown below:

```
state vars U: set of USERS
          O: set of OBJECTS
          L: POSet of LABELS immutable
          UM:set of ⟨u : USERS; l : LABELS⟩
          OM:set of ⟨o : OBJECTS; l : LABELS⟩
```

transitions

```
Read(user, obj) ∧ ⟨user, labeluser⟩ ∈ UM
∧ ⟨obj, labelobj⟩ ∈ OM ∧ (labeluser ≥ labelobj) → skip
```

```
Write(user, obj) ∧ ⟨user, luser⟩ ∈ UM
∧ ⟨obj, lobj⟩ ∈ OM ∧ (labeluser ≤ labelobj) → skip
```

To enforce this policy, immutable labels are added to all objects. User and Object labels can only be changed by the system owner. Before any read or write operation (e.g., before a send or receive), the labels are checked to see if they violate the two safety properties “no read up” and “no write down”, expressed as:

$$\boxed{\square(\text{Read}(\text{user}, \text{obj}) \Leftrightarrow (\text{label}_{\text{user}} \geq \text{label}_{\text{object}}))}$$

$$\boxed{\square(\text{Write}(\text{user}, \text{obj}) \Leftrightarrow (\text{label}_{\text{user}} \leq \text{label}_{\text{obj}}))}$$

To enforce this, all objects that send and receive information between each other have to be augmented with a guard that enforces this property. Dynamic policies in this

case will correspond to the policies that can add users and objects along with their labels.

However, other information flow policies can be expressed as liveness properties. Lamport [16] showed that the liveness property is dependent on the safety properties of sharing mechanisms. Liveness does not imply safety and vice-versa. A classical example is the specification of reliable streaming protocol. A property specification for such a protocol is given as (for all messages):

$$\boxed{\square(\text{send} \longrightarrow \diamond \text{receive})}$$

Here the \diamond operator stands for the *eventually* operator or the F (in the future) operator in LTL. This property implies that all messages sent must be eventually received. A simple windowing protocol satisfies this specification. The enforcer in this case is the windowing implementation, and dynamic policies to change the window size and parameters can change the operational parameters of the protocol without changing its behavioral guarantees. Liveness properties can be specified using the \square and \diamond operators [32]. However, as we see in the next subsection, this property specification does not address fairness and denial of service issues.

5.2. Availability Policies

In the previous subsection, we showed example specifications of liveness properties. However these policies do not impose any bounds on the availability of resources. A sender may experience starvation and never be able to send, in which case channel liveness as specified by the property is vacuously true at all times (recall that an implication $p \rightarrow q$ is true when p is false).

Now consider the following property:

$$\boxed{(\square \diamond \text{send} \longrightarrow \square \diamond \text{receive})}$$

This represents the behavior of a system in which, for all users, if the trace of system behavior contains sends infinitely often, represented by $\square \diamond$, then it also contains receives infinitely often. This says that if the sender does not starve, the message will be eventually received. While this property is adequate to represent the notion of fairness, it does not express availability constraints. Specifically, it does not impose any bounds on the amount of time (finite or minimum) a sender should wait before a user can send in the first place. A policy implementation that ensures availability for sends (or prevents sender starvation) would therefore include some bounds on individual resource consumption, e.g., and enforce this using channel arbitration, priority queues or other fair queuing disciplines that multiplex the senders packets. This ensures that if users want to send, they do not wait forever.

We also observe that the example fairness specifications do not prevent denial of service. What is missing from the specification is the notion of “making progress” or resilience to denial of service. Yu and Gligor [35] develop a Finite Waiting Time (FWT) Policy, to prevent denial of service and show that in order to model the notion of denial of service resistance, in addition to fairness, simultaneity and user agreements are required to make progress. In the channel modeling example, in order to prevent a user from waiting forever, several things need to happen at the same time (simultaneity). When the resource becomes available, the user must have something to send. It is not enough if only one of these conditions hold, to make progress. Denial of service can also take place because another high priority user may decide to send, delaying the send of the lower priority user. Therefore, in order to guarantee a finite waiting time, user agreements to preempt this type of behavior are also required. Millen [25] extends the notion of denial of service resistance by defining a resource allocation model that satisfies MWT or maximum waiting time policies in addition to FWT policies.

We are extending this notion and modeling the Distributed Denial of Service (DDOS) problem, by identifying appropriate property specifications and enforcers, and developing appropriate dynamic policies to change operational parameters while guaranteeing DDOS resilient behavior properties. We have developed an enforcement protocol using credentials that authorizes the use of bandwidth (CABs), along with a dynamic filtering implementation that enforces the user agreements, simultaneity, liveness, and safety properties required to prevent denial of service. The complete details of these mechanisms will be published in a forthcoming paper.

6. Related Work

In this section we present a brief summary of related work. Policy specification, reasoning and trust management are mature areas of research, and to include all related research is beyond the scope of this paper. We only attempt to highlight relevant recent research and any omissions are inadvertent. We classify related research into the following categories: security issues in active networks, enforceable policies, security policy specification, specification of access control policies, and trust management.

The Active Networks Security Working group, which includes the PIs in the Seraphim project, has developed a Security Architecture draft that highlights the important security issues, and a comprehensive threat and trust model for the active networking paradigm. Murphy et al.’s [28] proposal for strong security in active networks discusses the issues and requirements for authentication and authorization mechanisms in the active networking paradigm. This

research deals with the trust assumptions and protection mechanisms required to prevent different active networking entities such as the end-user, NodeOS, EE and active capsules from behaving maliciously and compromising the network infrastructure as a result. Parallel research in the Seraphim group [19, 22, 20] with respect to the use of standard APIs to augment this required security, together with a flow analysis of the security protocols enabled by the APIs, complement our research in dynamic policies. A secure active network infrastructure, though orthogonal to the work presented here, is a prerequisite to the deployment of dynamic policies in active networks.

The PLAN project [14] has developed a “Packet Language for Active Networks” which is a resource-bounded functional programming language. The fundamental construct in the language is remote evaluation of delayed functional applications. By restricting the language, PLAN allows users to develop active capsules that have attack resilience properties, such as CPU and memory denial of service protection, and guaranteed termination. The choice of a programming language to encode dynamic policies is important. In addition to the behavioral guarantees, programming language safety (such as PLAN) needs to be an integral part of any dynamic policy implementation.

Schneider defines a class of policies called Enforceable Policies [30] that can be enforced by execution monitoring. This class of policies is specified using a special automata called Security Automata and is concerned with the preservation of safety properties. Our class of dynamic policies super-scribes this definition by providing a general method for building security properties with verifiable properties, based on the modeling of system behavior and validation of property satisfaction. As such, enforceable policies are an important subset of dynamic policies.

Different notations and languages for security policy specification have been proposed by various researchers. These include the policy specification from the IETF Policy Framework Working Group [33], the original SPKI project, and the SecPol project. The IETF Policy Framework Working Group[33] is working on a policy framework specification that focuses on the development and enforcement of policies for QoS and IPsec applications. A special language to specify policy rules, which consist of a set of conditions and a set of actions, is proposed by this working group.

The SPKI system was proposed to provide mechanisms to support security in a wide range of Internet applications, including IPSEC protocols, encrypted electronic mail, WWW documents and payment protocols etc [13]. The SecPol approach advocates the use of a role-based framework to manage security in large, multi-organizational distributed systems[31]. The SecPol project has developed Ponder[11], a declarative, object-oriented language to specify security policies, group them into roles and relation-

ships, and define management structures. However, none of these projects include a formal modeling of systems for which the policies are developed, or use model checking to verify that the model can actually enforce the properties and policies of interest.

Jajodia et al. [15] propose a language for expressing authorizations and enabling the enforcement of multiple access control policies. They show how programs written in this language effectively capture the abstractions necessary to define different access control models. The security properties of implementation programs are not modeled.

Weeks [34] provides a formal semantics for expressing trust management systems via a fixpoint lattice model for monotonic assertions. This model is useful to understand the trust management of capability-based assertions. Chander et al. [8] provide a state transition approach to model the interaction of trust management and access control. The interaction of access control and trust management including the use of unforgeable credentials to provide authorization proofs and the equivalence of ACLs and capability lists can be validated in their framework.

The capability-based KeyNote [4] system of Blaze et al., provides a single language for both policies and credentials, based on predicates that describe the trusted actions permitted by holders of specific public keys (or other cryptographic identifiers). Our model integrates access control policy management with a simple trust management mechanism. The main purpose of KeyNote is to express and evaluate policies and trust delegations that occur in PKI applications. KeyNote can be integrated into our framework for trust management for other types of dynamic policies that require more expressive credentials.

7. Conclusions

To summarize, in this paper, we describe a general method to construct a special class of security policies we call dynamic policies. Through the policy development life-cycle, we explore the specification, verification, and validation of these policies using suitable formal notations and methods. We also describe the mechanisms for policy enforcement based on the implementation of the specification. The policies created in this framework can be updated by administrators during the operation of the system without compromising the security properties of the implementation. As an example, we develop a formal specification of access control policies as safety properties in the behavioral model of our system, whose enforcement is augmented with proofs of authorization. We also describe our implementation of dynamic access control policies in Seraphim, in the context of active networks, and demonstrate the power of our policies with two examples. Construction of other types of dynamic policies based on liveness, fairness and denial of

service resistance are also briefly described.

At a higher level, our research explores the behavioral descriptions of programs that can be sent across networks to change a system's software state. We explore this in the context of security guarantees that can be made about the system state before, during, and after the execution of such programs. This research is crucial in the context of active networking and in other dynamic environments where operational parameters are constantly changing. Our dynamic policy development life-cycle enables the creation of customizable programs that can be deployed on-the-fly to enforce and implement strong security policies. We also strongly believe that security concerns need to be integrated into models of system behavior, and security properties have to form an integral part of system specifications. Our major contribution is that we present a powerful set of methods and mechanisms that can be used to create policies with strong security guarantees, eliminating guesswork in the design and deployment of reactive security systems.

At a more fundamental level, we also argue that dynamic environments require dynamic security solutions. Dynamic policies enable administrators to react to vulnerabilities detected by IDS and risk analyzers with greater confidence. By including temporal properties in our design of security policies, we can change our system implementations in a controlled manner, and turn on restrictive attack resilient policies at will, without sacrificing security guarantees. This dynamism also allows us to change back to default policies after the attack has been mitigated, allowing us to implement minimal security solutions on a need to protect basis, and amortize performance penalties. We believe that this is our unique contribution in the context of other important research in security for active networks that explore the mechanisms needed to secure the underlying active network infrastructure, language safety issues, and secure bootstrapping etc. One of the major concerns in the active networking paradigm is how to change software state on routers "actively" without sacrificing protection guarantees. We believe that dynamic policies will be important components of solutions to address these concerns.

References

- [1] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. A domain and type enforcement UNIX prototype. In *Proceedings of the 5th Usenix UNIX Security Symposium*, Salt Lake City, Utah, June 1995.
- [2] D. Basin, M. Clavel, and J. Meseguer. "rewriting logic as a metalogical framework". In S. Kapoor and S. Prasad, editors, *Twentieth Conference on the Foundations of Software Technology and Theoretical Computer Science, New Delhi, India, December 13–15, 2000, Proceedings*, volume 1974, pages 55–80, 2000.

- [3] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, Bedford MA, 1973.
- [4] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust management for public-key infrastructures. In *Security Protocols International Workshop, Cambridge, England, 1998*.
- [5] J. Boyle et al. The COPS protocol. Internet Draft, February 24, 1999.
- [6] K. Calvert et al. Architectural framework for active networks. AN Architecture Working Group, Draft, 1998.
- [7] R. H. Campbell, Z. Liu, M. D. Mickunas, P. Naldurg, and S. Yi. Seraphim: dynamic interoperable security architecture for active networks. In *OPENARCH 2000*, Tel-Aviv, Israel, March 26–27, 2000.
- [8] A. Chander, D. Dean, and J. Mitchell. A state-transition model of trust management and access control. In *14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, June 2001.
- [9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
- [10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001. To appear.
- [11] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. Ponder: A language for specifying security and management policies for distributed systems. *Imperial College Research Report*, July 2000.
- [12] E. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
- [13] C. Ellison, B. Frantz, B. Lampson, R. Rivest, et al. SPKI certificate theory. RFC 2693, September 1999.
- [14] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998.
- [15] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26,2 of *SIGMOD Record*, pages 474–485, 1997.
- [16] L. Lamport. A simple approach to specifying concurrent systems. In *System Research Center, DEC, Palo Alto, CA, Report No., 15*, 1986.
- [17] L. Lamport. Specifying concurrent systems with TLA+. In *Calculational System Design*. M. Broy and R. Steinbrgen, editors, 1999.
- [18] L. Lamport. A formal basis for the specification of concurrent systems. Notes for the NATO Advanced Study Institute, June 2000.
- [19] Z. Liu. *Securing the Node of an Active Network*. PhD thesis, University of Illinois, Department of Computer Science, Dec. 2001.
- [20] Z. Liu, R. H. Campbell, and M. D. Mickunas. Securing the node of an active network. In *Active Middleware Services*. Kluwer Academic Publishers, Boston, Massachusetts, September, 2000.
- [21] Z. Liu, R. H. Campbell, S. K. Varadarajan, P. Naldurg, S. Yi, and M. D. Mickunas. Flexible secure multicasting in active networks. In *ICDCS International Workshop on Group Computation and Communication, Taipei, Taiwan, April, 2000*.
- [22] Z. Liu, P. Naldurg, S. Yi, R. H. Campbell, and M. D. Mickunas. Pluggable active security for active networks. In *International Conference on Parallel and Distributed Computing and Systems (PDCS 2000)*, Las Vegas, Nevada, November 6-9, 2000.
- [23] Z. Liu, P. Naldurg, S. Yi, T. Qian, R. H. Campbell, and M. D. Mickunas. An agent based architecture for supporting application level security. In *DARPA Information Survivability Conference and Exposition*, Hilton Head Island, SC, January 25-27, 2000.
- [24] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track of the 2001 USENIX Annual Technical Conference*.
- [25] J. K. Millen. A resource allocation model for denial of service. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 137–147, 1992.
- [26] R. Mundy, D. Partain, and B. Stewart. Introduction to SNMPv3. RFC 2570, April 1999.
- [27] S. Murphy et al. Security architecture for active nets. AN Security Working Group, July 15, 1998.
- [28] S. Murphy, E. Lewis, R. Puga, R. Watson, and R. Yee. Strong security for active networks. In *2001 IEEE Open Architectures and Network Programming Proceedings (OpenArch 2001)*, Anchorage, AL, Apr 27-28, 2001. pp 63-70.
- [29] P. Naldurg and R. Campbell. Dynamic access control policies in seraphim. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 2002.
- [30] F. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
- [31] SecPol. SecPol project homepage, 2000. URL: <http://www-dse.doc.ic.ac.uk/projects/secpol/SecPol-overview.html>.
- [32] A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing* 6(5): 495-512 (1994), updated 1999.
- [33] M. Stevens et al. Policy framework. IETF draft, September 1999.
- [34] S. Weeks. Understanding trust management systems. In *2001 IEEE Symposium on Security and Privacy*, May 2001.
- [35] C.-F. Yu and V. D. Gligor. A formal specification and verification method for the prevention of denial of service. In *Proc. 1988 IEEE Symposium on Security and Privacy (Oakland '88)*, Oakland, CA, USA, Apr. 1988, pp.187-202.