SECURITY POLICY CONSISTENCY AND DISTRIBUTED EVALUATION IN HETEROGENEOUS ENVIRONMENTS

Sotiris Ioannidis

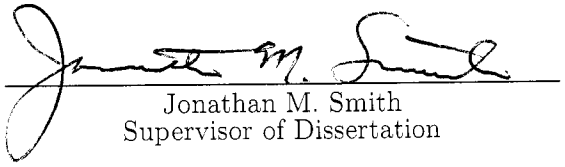
A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2005



Jonathan M. Smith
Supervisor of Dissertation

Rajeev Alur
Graduate Group Chairperson

Acknowledgements

I would like to start by thanking my advisor and committee members. Jonathan Smith, for being much more than an advisor and never losing faith in me. It has been a strange trip indeed, thanks JMS! Michael Greenwald, for all our technical and non-technical discussions. Matt Blaze, for getting me past the finish line. Steven Bellovin, for giving me the opportunity to work with him. Finally, Honghui Lu, for saving the day by being on my committee.

The Greeks and the Barbarians of DSL, Kostas Anagnostakis, Angelos Keromytis, Vassilis Prevelakis, Eric Cronin, Michael Hicks, Stefan Miltchev, and Jonathan Moore.

The friends I made in Rochester, Amanda Stent and Christopher Eveland.

The administrative staff of the department, Cheryl Hickey, Michael Felker, Gail Shannon, and Mark West, for making my UPenn life so much easier.

Last but not least, my family and Vicky Danilatou, they all stood by me throughout this journey. I hope we have many more good years together.

This work was supported by Air Force Research Lab (AFRL) "STRONGMAN: Scalable Trust of Next Generation Management" under contract F39502-99-1-0512 and NSF

"GRIDLOCK: A New, Scalabe Approach to Unifying Computer and Communication Security" under contract CCR-02-08972.

ABSTRACT

SECURITY POLICY CONSISTENCY AND DISTRIBUTED EVALUATION IN HETEROGENEOUS ENVIRONMENTS

Sotiris Ioannidis

Jonathan M. Smith

Modern computing environments involve a multitude of components working in concert to provide services to users. Computer elements, network elements, operating systems, applications, users, *etc.* must to be constantly managed and controlled to prevent unintended or illegal accesses that could compromise the system. Such systems rely on a security policy, namely a set of rules that define what actions are allowed and disallowed to be performed in the system, defined by policy makers to control its behavior. To avoid failures (*i*) information about the state of the system must be readily available to the enforcement elements, for them to make decisions according to the specified security policy, and at the same time (*ii*) the security policy itself must be maintained in a consistent state across the multitude of system elements.

This dissertation addresses the problem of security policy consistency in decentralized heterogeneous systems. We present two novel ways for maintaining consistency in this type of environments. First, by using both *static* (compile time) and *dynamic* (run time) techniques, in which security policies are examined against each other and possible inconsistencies are identified. Second, we demonstrate how it is possible for multiple security elements to dynamically *share* and *exchange* state information, to consistently enforce security policies that span multiple access control nodes.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Security in computer systems is engaged with protecting resources from unauthorized access. Of course, to achieve security in computer systems one has to address a multitude of issues, like software correctness, authentication, authorization, configuration, *etc.* This work focuses on security policy. Security policy is the collection of rules that express what actions are permitted or disallowed to happen in a computer systems [Lam71, Slo94, SV00]. More specifically, this thesis recognizes that there are two very important problems concerning security policy in modern computing environments:

1. In heterogeneous systems there is a semantic gap between the security policy specification level and the policy enforcement level.

2. In decentralized systems there is limited cooperation in security policy enforcement between access control nodes.

1.1 The Problem

Securing single computers has proved to be a very difficult task.[1] Securing large numbers of computer systems can be an even more challenging process. Complex computing environments have become the norm in modern organizations. In these environments, computers, along with other devices *e.g.*, printers, network attached storage, thin clients, *etc.*, are interconnected by computer networks which are themselves extremely complex. These networks normally contain a variety of network elements, such as routers, switches, and firewalls, forming complex topologies. The hardware configuration, however, is only part of the complexity equation. The difficulty is further increased by the diversity of operating systems running on these devices, along with the plurality of application software. Finally, the existence of multiple administrative domains, as it is often the case (network administrators, web administrators, *etc.*), can introduce anomalies in the security policy of the organization.

Let us examine how these issues complicate the task of security policy management and enforcement in complex, decentralized organizations. The growth of computer systems in scale (number of processors, speed, geographic distribution, *etc.*) and in complexity (heterogeneity, range of applications, complexity of implementation, *etc.*) only increases the difficulty of administration. This difficulty is especially apparent in the specification and enforcement of security policies in distributed systems. While centralized and (mostly) homogeneous systems have the benefit of simplified management due to controlled actions being at a single place, such systems are evolving into distributed and heterogeneous

[1] Just by looking at the CERT advisories [cer] we realize the multitude of vulnerabilities that exist in modern computer systems.

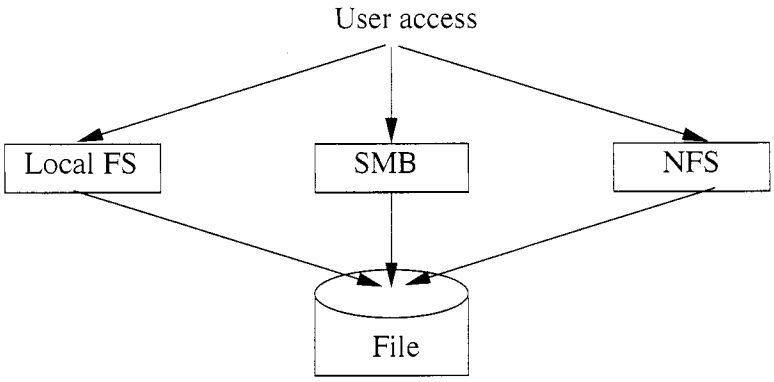


Figure 1.1: **A user file access should be subject to the same security policy independently of where the request is issued. If the user is not permitted to access a file on the local file system, they should not be permitted to gain access via NFS of SMB.**

environments. Large numbers of resources can no longer be control using a centralized approach [IKBS00, Ker01]. Management and control of diverse users, applications and objects has become unsustainable in such environments [Mar97]. Also, assuming that one manages to correctly configure such a system it is inevitable that over time, as applications change, users come and go, topologies are modified, *etc.*, it will reach an inconsistent state. Tools that will help users manage computer systems and support a *consistent* specification of security policy in heterogeneous distributed environments are becoming crucial.

The term consistent is overloaded with many meanings. Informally, we say that a security policy is enforced consistently, if each policy rule is enforced the same way everywhere in the system. We will give more formal definitions in Section 1.3.1 and Chapter 3. For now, let us look at a simple example to illustrate our point. Consider a rule forbidding a user from accessing a certain file. Such a rule should be honored no matter how or where that user attempts to gain access to the file, locally, over NFS, or via SMB (see Figure 1.1).

For the security policy to be consistent, the policy evaluation, that is the actual interpretations of a policy rule when it gets triggered by a user action, has to be the same everywhere. In our file access example, if a user is not permitted to access a file of the local file system, they should not be allowed to access it when the same file is mounted on a remote machine. Additionally, we talk about security policy consistency in the event of possible policy conflicts. In this case we require the presence of a deterministic conflict resolution protocol that will resolve the contradictions. Manual or semi-automatic configuration of nodes and protocols to conform to a global policy has been shown to be problematic and error-prone [How97], therefore automatic techniques relying on a single method of specification are desirable.

However, prior work has mostly focused in the area of policy conflicts [JSS97, LS97, CC97, LS99], policy specification [BFIK99b, DDLS01, RZFG01], and distributed enforcement [TJM+99, KIGS03b, KIGS03a].

It is the goal of this dissertation to help automate many of the tasks that are required for management and enforcement of a consistent global security policy in a decentralized heterogeneous system. Considering the issues we have mentioned earlier the difficulty is twofold.

The first challenge is associating high-level/organization-level policies all they way down to the enforcement layer (see Figure 1.2), in such a way as to avoid policy inconsistencies. For example an organization policy that states: "only trusted users should connect to trusted machines," must be translated into the appropriate rules in the various enforcement elements, and enforced they *same* way on each and every one of them. The second challenge is making sure that all the enforcement elements operate together to implement this type

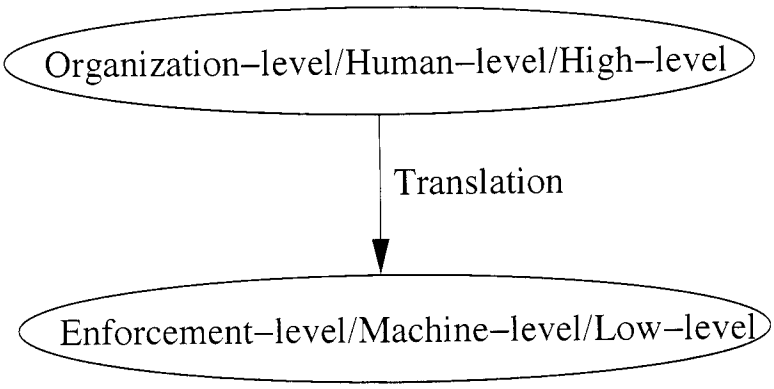


Figure 1.2: **Abstract security policies must be translated to specific policy rules that enforcement nodes can interpret.**

of organization-level policies.

1.2 Background and Motivation

Lampson [Lam71] describes the properties of protection mechanisms, as well as how such mechanisms can be implemented and used. He further argues that even though a system can be complete without implementing access control, for example in the case where all system users are both benign and infallible, in reality users make mistakes and are not always benign.

It is therefore necessary to build systems that incorporate mechanisms for access control. All modern operating systems provide mechanisms for accomplishing access control, most notably, permission bits, access control lists, and packet filters. Access control *policy*, however, as opposed to access control *mechanisms*, involves *"high-level"* rules that govern subjects [SV00]. So even though in its simplest form a security policy can be expressed with very basic primitives, such as permission bits or access control lists, many systems

lack the concept of real *"high-level"* policy specification.

There are a number of difficulties that arise from using basic primitives to define security policy.

1. The access control primitives are too low-level and implementation specific, exposing too much detail about the underlying system [WL93]. It is therefore desirable for authorization statements to be in a form independent of the implementation mechanism.

2. Since systems tend to change over time, users come and go, hardware becomes obsolete and gets replaced, network topologies change, *etc.* [BCG+01, KFJ03].

3. Finally, the policy itself might be too complex to be expressed by simple primitives, *e.g.,* security policy that changes depending on the time of day, *etc.* [CL98].

To address these difficulties a number of policy specification languages have been proposed over the last several years like the Security Policy Language (SPL) [RZFG00], Ponder [DDLS01], and XACML [GM03]. These languages address the above issues by offering an abstraction layer over the low-level access control primitives. Access requests and events are propagated into their runtime system where policy is evaluated. Additionally their syntax is rich enough to express complex policies often inexpressible by the typical access control primitives.

Even though policy languages simplify system management and assist users in setting up security policy it is possible to introduce policy inconsistencies. Policy inconsistencies come in the form of policy conflicts. More specifically modality conflicts, that is, a positive authorization conflicting with a negative authorization.

Of course such conflicts are not unique to security policy rules. Postgres [SHP88, PS96] for example assigns explicit priorities to rules. Jajodia *et al.* in [JSS97] consider cases where the positive authorization takes precedence, and where the negative authorization takes precedence. Systems like KeyNote [BFIK99b] only allow positive authorization, avoiding modality conflicts in policy rules. In [LFG99] the authors discuss conflict resolution by rule prioritization. They offer three criteria as the basis for rule priority: *specificity*, more specific rules override more general ones, *authority*, higher authority takes precedence of lower authority, and finally *recency*, in which more recent rules take precedence over older rules. Larrondo–Petrie *et al.* in [LPGSF90] suggests the use of *distance* between a policy and the managed object as a measure of the applicability of a rule.

This work will not investigate the above problem, namely the introduction of inconsistencies in the security policy due to conflicts. Instead what we are interested in is consistency with respect to *distributed policy evaluation.* The problem is two-fold, since we are addressing distributed heterogeneous systems, the same high-level policy must *mean* the same thing across different platforms, and also it is possible for parts of the security policy to be evaluated on multiple different nodes.

To address the first issue we must be able to provide some assurances that the security policy is evaluated the same way on different platforms. As for the second problem, these nodes in the system will have to work together to enforce the global security policy. To our knowledge there is no prior work on this subject. A number of systems, like Akenti [TJM+99] and the Distributed Firewall [IKBS00], decouple policy specification from policy enforcement. Specifically, security policy is encoded in cryptographic credentials and distributed to the enforcing nodes. We propose something more powerful, expressing security

policy that spans multiple nodes. For example in our system it would be possible to define and evaluate policy rules that depend on state resident on multiple system nodes. We will expand on these problems in Chapter 3.

1.3 High-level Security Policy Specifications

Most organizations delegate authority hierarchically: the CEO or Chief Security Officer can set company-wide policies, and subordinates may only craft policies that conform to the corporate policy. Similarly, department heads refine the corporate policy on a per-department basis, restricting the policies under control of group leaders, and ultimately, the actions of individuals. It is important to note that the hierarchy of authority is not unique; rather it is relative to an object in question. While in general the management or organizational structure induces a hierarchy of authority (*i.e.,* typically CEO's rules override department rules, that override individuals rules, *etc.*), in fact the relative priority of authority may shift. This occurs, for example, when someone is, say, on special assignment, or when the object in question is a personal file owned by a newly hired employee at the bottom of the hierarchy — who may have greater authority over that particular file than his manager does.

In *any* security system one can approximate this hierarchy and delegation by adopting the "social model" (and in most systems this is the best that we do). In the social model, we assume an out-of-band mechanism for specifying security policies (an employee security handbook, for example) along with a company rule that requires employees to follow these policies. It is clear that at the level of individual file permissions, the "social model" is

untenable and will inevitably break down [And01]. Only in an ideal world will individual users set file permission on individual files in conformance with the policies set all the way up the authority hierarchy with only the help of a handbook. In the world of computer systems it seems clear that some automated enforcement is necessary if we want to ensure consistent policies.

Let us postulate the existence of a system that does enforce hierarchical policies all the way down to individual users and objects. That still begs the question of how a CEO actually specifies and manages policies that are meaningful at the level of objects (file or TCP port, for example). It is fair to say that few CEOs, Provosts, or Admirals (pace Bill Gates, John Hennessy, and Grace Hopper) know what ACLs or mode bits are, let alone how to `chmod 640 TestFile`. Rather, a more reasonable example of a corporate policy might be: "only employees and people who have signed Non-Disclosure Agreements (NDAs) may view confidential files". This implies a mapping of individual users into groups ("employees", "people who have signed NDAs"), objects into groups that are meaningful for a given application ("confidential files") and primitive operations for a given platform into higher-level operations that are meaningful to the user ("view"). These groups may be ephemeral, time-dependent, non-hierarchical, overlapping, or irregular in many other ways. The most appropriate terminology will almost always be application-dependent, so *the specification language must be extensible to cover new applications.* The implementation will be platform- and application-dependent, so *a means must be provided to implement the high-level terminology in platform- and application-dependent ways.*

We observe that people with domain expertise in a given application or platform are not necessarily (nor even likely!) the same people with authority over security policies in

that area. In other words, there are people who set policy and another group of people who translate the policy into working rules.

From the above discussion it is clear that many different high-level policies, each possibly specified by distinct administrative authorities, can be relevant to a specific instance of an attempted operation on a resource. We must determine which of these several policy rules applies. Further, it is clear that these high-level rules will have different translations, or implementations, for different applications on different platforms. How can we be sure that the rules are consistent with the intent of the human specifiers? How do we know that the rules are applied consistently on all platforms, given the different translations? How can we be sure that the myriad rules implement a coherent policy?

The short answer is: *"we don't know, and we can't be sure,"* since answering such a question would be equivalent to solving the halting problem. The longer answer is that despite the fact that it is impossible, in general, to know for sure, we can, in many cases, increase our confidence that the rules are "consistent."

1.3.1 Enforcing Security Policies Consistently

It is clear that we are speaking of at least two or three different kinds of consistency here. First, "consistency" can mean preservation of intent – that the implementation of the policy is consistent with the intentions of the administrators who specified the policy. Second, "consistency" can mean cross-platform consistency — enforcing the *same* rules across all platforms. If person P is not permitted to read a file on machine M, s/he should not be able to read that file by mounting it on machine N. Third, "consistency" can mean the absence of modality conflicts — that when several policies apply to a given action at

a given time, they all agree on whether to allow or deny the action.

Preservation of intent is important because it provides some assurance that the system obeys the rules set by the administrators. However, this is hard to guarantee. First, it is notoriously hard to write formal specifications, and there is no way of expressing the intent of the rule writers short of writing the rules. Second, even if the high-level rules are "correct", it may not be possible to directly map them to the available low-level access control mechanisms due to their potential limited expressiveness. Third, even if the rules are expressible in the low-level mechanisms available on each platform, the mapping from high-level specification to low-level rules may be incorrect. There are some limited things we can do to address the question of preservation of intent, and we discuss these later, but this work is primarily focused on the second meaning of consistency: the extent to which we can ensure consistent enforcement of the rules across all platforms.

Where the set of high-level rules reflect the intended policy (to the extent possible), the question of consistency across platforms will depend on the amount of control we have over low-level enforcement mechanisms. First, there are systems that fit the model outlined earlier, designed from the start with separable security mechanisms. Roughly speaking, on such systems, the security mechanism on *every* platform is an almost direct interpretation of the high-level policy rules by the access control points, *e.g.,* applications, compilers, operating systems, *etc.*. That is, we have total control of the access control points, and we can modify them and extend them to interpret our security policy. Second are systems where the security policy and/or its enforcement mechanisms are embedded in the system and the application. In such systems the best one can achieve is have the high-level rules map to configuration files and execute commands provided by off-the-shelf applications,

for example, filter rules in commercial firewalls or databases, or even setting file permission bits in a file system. In these cases we have to rely on the controls that are exported by each application.

There is very little we can do to guarantee consistency in the latter case, and we have to take on faith that the system does what it claims to do. We can, at best, ensure that the high-level policy mappings on each platform are as close to each other as practicable. The former case, where the system is an instance of a separable security architecture, is the subject of this work. It may seem trivial to maintain consistency when the access points enforce the same rules on every platform, but as we shall see, the mapping introduced by the adaptation layer raises potential problems. However, before we can directly address the question of consistency across platforms in our system, we need to introduce some terminology.

Any system is composed of collections of agents and resources in the form of users, files, printers, network elements, *etc.* Most of these objects have some existence — and a notion of object identity — outside of the abstractions of the operating systems and applications that use them. We will refer to these as *low-level objects* (LLO). High-level policies in a heterogeneous system mediate generic operations on LLO's. If the policies are specified in a high-level security language, the LLOs will be identified through identifiers or descriptors in that language. Thus, in any high-level security language there is a mapping from abstract, high-level, identifiers to LLOs. For convenience we will refer to the abstract representations of these objects as *high-level objects* (HLO). Note that there is not always a direct, one-to-one, mapping between HLOs and LLOs. For example, we are likely to talk about `TRUSTED_USERS` rather than specific users (*e.g.*, `root`), or `SECURE_FILES` rather than

Policy Language Space	HLO	`TRUSTED_USERS, SECURE_FILES, ...`
Application/Operating System Space	ILO	`root, /etc/passwd, C:somefile, ...`
Physical Reality Space	LLO	Actual resources, users, files, *etc.*

Table 1.1: **We define three layers of objects: high-level objects used by the policy language, low-level objects which refer to actual resources, and intermediate-level objects which represent how the applications and operating systems view them.**

listing files such as `/etc/passwd`. Further, the same LLO may be in the image of more than one HLO,[2] and HLOs may refer to infinite sets, some of whose members do not yet exist (*e.g.,* dynamically created groups).

However, it is operating systems and applications that ultimately mediate access to LLOs. These have their own abstractions for LLOs. For example, a file name, or file descriptor, or inode, is an abstraction of the bits on a disk. Additionally, different application or operating systems might have different views of an identical LLO. For example the same file system might be mounted on a Unix box via NFS and on a Microsoft Windows box via SMB, or the same user might be known as `foobar` on Linux and as `Foo Bar` on Mac OS. We refer to these, existing, abstractions of LLOs as *intermediate-level objects* (ILO). We summarize these object layers in Table 1.1. The ILO-LLO mapping is also not usually one-to-one; it may be one-to-many (file system groups, buddy lists), or may be inconsistent across platforms (*e.g.,* root, Administrator). Figure 1.3 illustrates the three kinds of object layers along with the mappings.

An adaptation layer between the high-level policy language and specific components of the system can be viewed, roughly, as a mapping between HLOs and ILOs. This is not a simple mapping, because HLOs can refer to infinite sets, or sets of objects that do not

[2]In such a case, although contradictory rules may apply, we have an algorithm (described below) that uniquely determines whether to grant or deny access in each particular instance.

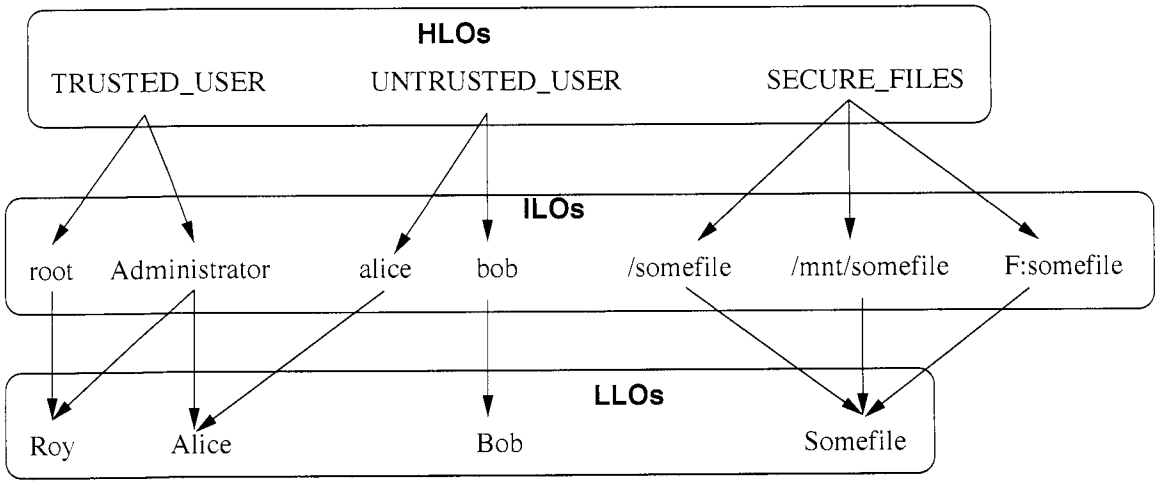


Figure 1.3: **Mappings of HLOs to ILOs and ILOs to LLOs.**

yet exist. The adaptation layer may provide a simple translation between an HLO and an ILO, or it may map an HLO that refers to a set to a corresponding ILO that also refers to a set, or it may identify an HLO with a predicate over ILOs.

With this terminology in place we can reformulate the questions of intent preservation and consistency. To determine whether a system preserves intent, we must answer two questions: First, whether, given the mapping of HLOs to LLOs, the set of rules applying to a specific operation applied by a particular user to a given object at a certain time gives the intended result. Second, whether the composition of the HLO-ILO mapping with each of the ILO-LLO mappings preserves the intended HLO-LLO mapping.

In a system with separable security mechanisms, the question of consistency across platforms boils down to asking whether the composition of HLO-ILO, ILO-LLO mappings maintains the same HLO-LLO mappings for all platforms in the system.

We can express this slightly more formally. Let $\mathcal{H}$ denote the set of HLOs, and $\mathcal{L}$ denote the set of LLOs. Let M be the HLO-LLO map for the system, whose type is $\mathcal{H} \times \mathcal{L}$.

That is, M determines whether a given LLO is a member of the set of objects referred to by a given HLO.

For each platform p type, let H_p and L_p, respectively, refer to the HLO-ILO and ILO-LLO "mappings" for p. Unfortunately, these mappings may be peculiar in many ways, may be defined implicitly, or may refer to non-existent objects. In particular, it is hard to know how to define the type of ILOs, and therefore the type or specification of H_p and L_p. Luckily, we need never consider H_p and L_p in isolation, but only their composition, denoted by I_p. I_p shares the type of M, and regardless of the messiness at the level of ILOs, I_p maps HLOs to sets of LLOs. It is tempting to view I_p very loosely as the implementation of the high-level policy that is (incompletely) specified by the combination of M and the set of high level policy rules. In such a view, the question of consistency could be settled by asking whether, for all p, I_p meets the specification M. However, we cannot. In the normal course of events there is no explicit specification of M. If M were actually specified in some way (formal, or not), it would still not be completely sufficient to determine consistency because, among other difficulties, there are LLOs whose membership in an HLO is unknown, as well as LLOs in multiple HLOs.

We summarize our definitions:

Def 1: Let $\mathcal{H}$ denote the set of HLOs, and $\mathcal{L}$ denote the set of LLOs. We define a mapping $M : \mathcal{H} \rightarrow \mathcal{L}$ in $\mathcal{H} \times \mathcal{L}$ which determines whether a given LLO is a member of the set of objects referred to by a given HLO.

Def 2: Let p be some platform. Let $\mathcal{H}$ denote the set of HLOs, and $\mathcal{I}$ denote the set of ILOs. We define $H_p : \mathcal{H} \rightarrow \mathcal{I}$ in $\mathcal{H} \times \mathcal{I}$ as the mapping of HLO-ILO for platform p.

Def 3: Let p be some platform. Let $\mathcal{I}$ denote the set of ILOs, and $\mathcal{L}$ denote the set of LLOs. We define $L_p : \mathcal{I} \rightarrow \mathcal{L}$ in $\mathcal{I} \times \mathcal{L}$ as the mapping of ILO-LLO for platform p.

Def 4: Let $\mathcal{H}$ denote the set of HLOs, and $\mathcal{L}$ denote the set of LLOs. $\forall$ platform p, we define $I_p = H_p \circ L_p$ the composition of H_p and L_p in $\mathcal{H} \times \mathcal{L}$, which maps HLO to sets of LLOs.

Def 5: A policy is consistent *iff* $\forall$ platform p, $M \equiv I_p$, meaning the two mapping specifications are equivalent.

Although we cannot formally check whether M is equivalent to I_p,[3] the notion of treating M as a partial specification and testing I_p against it can be used to give us a general method for increasing our confidence in the consistency of the system. We will see that as long as a name system exists for LLOs, and as long as policy writers are willing to specify M, at least partially, then we can check I_p against M to some extent.[4] In some cases — where M is relatively complete and I_p is trivial — cross-platform consistency can be determined completely. But in most cases, we can only statically determine to a certain degree whether cross-platform consistency is met.

1.4 Goals and Novelty

To recapitulate, this work will:

1. Demonstrate that it is possible for multiple security elements to work together to enforce the global security policy, permitting more expressive policies. This is achieved

[3] Remember that M might not be completely specified.
[4] The extent is a function of the completeness of the specification of M.

by enabling multiple security elements to dynamically query each others state. This *sharing* and *exchange of* information enables administrators to write policy rules that cross enforcement layers and nodes.

Past security languages/architectures (*e.g.*, [DDLS01, RZFG01]) assume knowledge of who has direct access to all system state, either by assuming it is locally available or by assuming a common repository. Such assumptions are not realistic in today's computing environments. In reality organizations are comprised of many different applications, operating systems, devices, networks, and users, each with only local knowledge and representation of state.

2. (a) Determine the set of conditions that are required to *statically* guarantee policy consistency on heterogeneous environments. That is, systems that are comprised of different kind of computing elements, networks, operating systems, applications, *etc.*

   (b) Demonstrate that it is possible to *dynamically* check for consistency at runtime.

   Past work focuses on detecting/resolving modality conflicts on systems that use a common, static, object namespace. Our work extends this to address different platforms, each with its own namespace, aliasing, and dynamically generated objects.

3. Demonstrate the advantages of *late-binding* in policy specification. By late-binding, in this context, we mean the ability to write rules and reason about them, about managed objects that might not exist at the time the rule is specified.

   It is theoretically possible to design a security policy specification language in which *all* rules can be statically checked for consistency at specification (or "compile time").

However, we chose to extend the expressiveness of our language — sacrificing the chance for static consistency checking — for two reasons.

(a) First, some policies seem impossible to express in a language that is statically checked for consistency. For example, rules that refer to groups whose members are dynamically created objects, and whose membership may differ over time and over different platforms.

More specifically, consider the case of user processes that are permitted to access a network printer. Processes are transient objects and users have different identities on different hosts. To check such policy rules for consistency statically would be extremely difficult, since we don't even know whether an object exists or not. A runtime approach however would greatly simplify things as we can check for policy rule consistency as objects are created.

(b) Second, even for rules that are expressible in a more restricted but statically checkable language, I believe that they can be expressed more economically and clearly in a more dynamic language, making it more likely that the rules match the author's intent.

Consider again the previous example. One could conceivably list all possible enumerations of processes and users on all possible hosts, and write policy rules to control them. However, such an approach would be time consuming and tedious, and hence error-prone.

Specifically this work created a framework that permits *consistent* high-level security policy definition and achieves *cooperative* policy specification and enforcement on

distributed heterogeneous systems. By cooperative we mean that different enforcement components work together to implement the defined security policy. Such a system will avoid policy inconsistencies, and therefore security failures, that are often common on distributed security architectures.

The approach we take for policy definition is to apply the abstraction refinement model of compilation to policy specifications, automatically translating them to policy enforcement. Each stage of abstraction refinement results in a more localized and specialized representation of the policy, much as a high-level programming language is gradually refined into machine code for a specific architecture. We see this as a way to achieve succinct and portable representations of system security and access control policies, reducing errors and increasing heterogeneity in the enforcement mechanisms. As with modern programming language systems, we see the basic approach resulting in a delegation of details to the mechanical translator, which can employ enforcement-mechanism specific details to make performance penalties negligible or even optimize the system.

Figure 1.4 illustrates our approach.

To assist users in their administration tasks and to avoid consistency problems it is important to centrally administer and specify each security policy rule, even if it applies to diverse types of objects over multiple enforcement points. Centralized administration helps avoid such consistency problems and eliminates the overheads of maintaining separate configurations across the nodes of a distributed system. Distributed enforcement of the security policy, on the other hand, aids scalability as the number of users, enforcement points, and rules grow. These observations imply that systems should combine centralized policy with decentralized enforcement, and, naturally, such systems are becoming

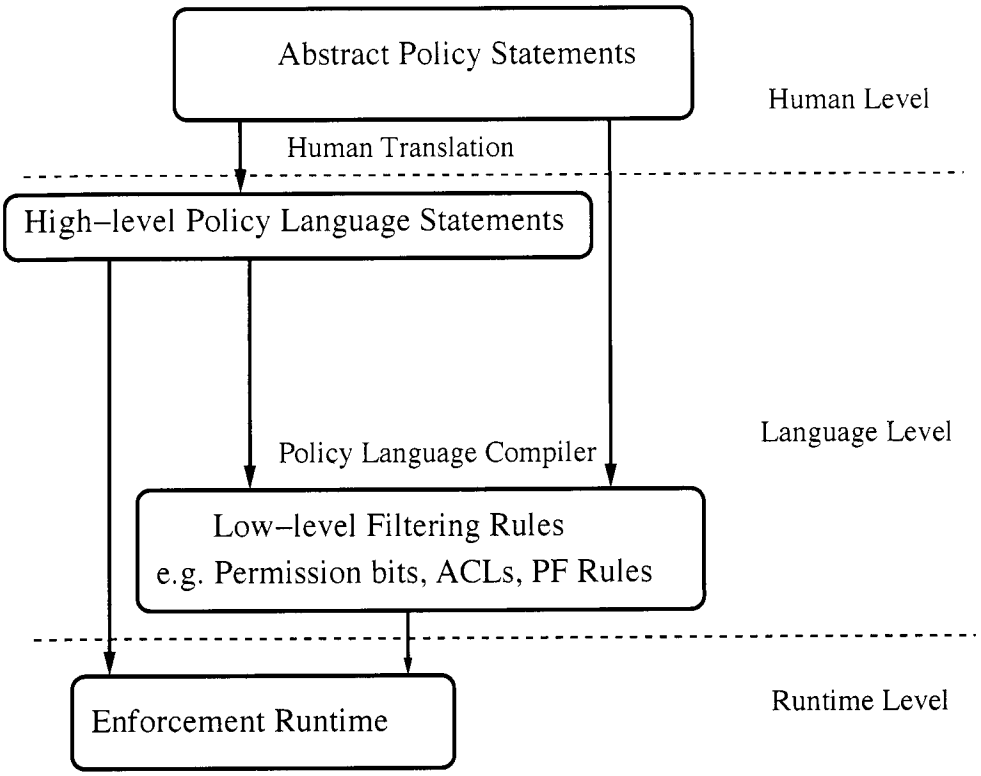


Figure 1.4: **Policy is defined at an abstract human-level and is gradually refined to a low-level specification for each type of enforcement element. Our goal is to guarantee that the high-level policy specification is consistently enforced across the entire system.**

increasingly common [IKBS00, BFIK99b, DDLS01].

1.5 Results and Contributions

I summarize the results and contributions of this dissertation:

1. The *first* system which provides a procedure that determines policy consistency given a high-level security policy rules and a set of platform dependent mappings, (see Section 1.3.1 and Chapter 3).

2. The *first* system that utilizes multi-hop queries between enforcement elements permitting the exchange and sharing of policy state (see Chapter 4).

3. I demonstrate the existence of policies that cannot be checked for consistency statically at "compile-time" but can be checked dynamically at "run-time."

1.6 Thesis Organization

The rest of the thesis is organized as follows. In the next section I discuss previous work done in the area of policy specification and conflict resolution. In Chapter 3, I describe the problem of policy consistency in greater detail and present our solution for detecting policy inconsistencies. In Chapter 4, I address the problem of cooperation between policy enforcement elements. Chapter 5 describes two systems built while exploring the problem of policy consistency in decentralized heterogeneous systems. Finally, Chapter 6, contains concluding remarks and suggests future directions.

Chapter 2

Related Work

Computer system security has been researched extensively. However, demand for communication and openness have put new strains on computer systems. Communication environments such as the Internet requires us to solve a whole new set of problems (*e.g.*, large scale configuration and management of users, computers and services, distributed access control, *etc.*) that researchers have only recently began to address.

There are several methods for protecting computer systems, ranging from type-safe languages [LDOW98, MF97, WBDF97, Gon99], fault isolation [WLAG93] and code verification [NL97] to operating system-specific permission mechanisms [MK97] and system call interception [BBS95, GWTB96]. Furthermore users often rely on firewalls to isolate trusted parts of the network from untrusted ones. To configure such mechanisms researchers have developed a number of domain specific languages that assist users in specifying security policy.

2.1 Policy Specification

2.1.1 Trust Management

The Trust Management work at AT&T Labs Research [BFL96, BFIK99a, BFS98] developed a language for defining policies based on credentials and an engine that answers questions of the form: *"does request R, supported by credentials C, comply with policy P?"*, effectively binding authorization and authentication in a single credential. Their system does not address how the policy interacts with the rest of the system, or how it flows from one component to another. Also their system is monotonic and does not support negative authorization. Such an approach tends to cause an explosion to the policy state space. Having negative statements in the language, on the other hand helps in forming a compact security policy specification.

In [HMM+00], Herzberg *et al.* with their Trust Policy Language (TPL), use XML to specify trust policies and also permit negative authorization. As with the AT&T work, they cover specification and implementation of the authorization policy but don't address the policy life-cycle. In TPL clients are assigned roles using certificates, a role is a group of entities with some specific capabilities and permissions (*e.g.,* employees, managers, *etc.*). TPL uses X.509v3 certificates to identify subjects and also uses XML syntax to define policy rules.

Another trust-based system is Akenti [TJM+99, TMEC02, TEM03]. Akenti is very similar in concept to KeyNote and TPL: policy is written in XML and stored in three different kinds of certificates. *Policy certificates*, which specify the sources of authority of the resource, *use-condition certificates*, which contain the access control constraints,

and *attribute certificates*, which assign attributes to users. Akenti was designed for use in distributed environments and to simplify policy specification by taking advantage of XML. We feel that its use of X.509 certificates and TLS makes it easy to incorporate in web-like applications, however we believe that Akenti and TPL policy specification is both cumbersome and low-level due to the verbosity of XML (see Figure 2.1).

2.1.2 Higher-level Policy Languages

Ponder [DDLS01] is a language that provides a common means of specifying security policies that map onto various access control mechanisms for firewalls, operating systems, and Java. Ponder is declarative and object-oriented, which makes it fairly flexible. However it requires a lot of details to be specified by the policy administrator, and therefore fails to provide a significantly high-level security policy specification.

The Security Policy Language (SPL) [RZFG01] proposed by Ribeiro *et al.* is a language very similar to Ponder. It is able to express permission and prohibition, and some restricted forms of obligation. Since SPL policies are non-monotonic it is possible for conflicts to emerge. SPL assigns explicit priorities to policy rules to resolve them.

The Policy Description Language (PDL) is an event-driven language developed in the network computing research department of Bell-Labs [LBN99, VLK00, BLK00]. It uses the *event-condition-action* rule paradigm to define a policy as a function that maps a series of events into a set of actions. PDL does not support access control policies, nor does it support the composition of policy rules across different layers.

The Policy Group, within the IETF, focuses on quality of service management and configuration in networks. Their policy framework can be used for classifying packet flows

```
<Policy PolicyId="SamplePolicy" ...>
  <Target><Subjects><AnySubject/></Subjects>
    <Resources><ResourceMatch MatchId="...:string-equal">
    <AttributeValue DataType="..#string">SampleServer</AttributeValue>
    <ResourceAttributeDesignator DataType="...#string" ...>
    </ResourceMatch></Resources>
    <Actions><AnyAction/></Actions>
  </Target>

  <!-- Rule to see if we should allow the Subject to login -->
  <Rule RuleId="LoginRule" Effect="Permit">

  <!-- Only use this Rule if the action is login -->
  <Target><Subjects><AnySubject/></Subjects>
    <Resources><AnyResource/></Resources>
    <Actions><ActionMatch MatchId="...:string-equal">
    <AttributeValue DataType="...#string">login</AttributeValue>
    <ActionAttributeDesignator DataType="...#string" ...>
    </ActionMatch></Actions>
  </Target>

  <!-- Only allow logins from 9am to 5pm -->
  <Condition FunctionId="...:and">
    <Apply FunctionId="...:time-greater-than-or-equal"
    <Apply FunctionId="...:time-one-and-only">
    <EnvironmentAttributeSelector DataType="...#time"
      AttributeId="...:current-time"/></Apply>
    <AttributeValue DataType="...#time">09:00:00</AttributeValue>
    </Apply>
    <Apply FunctionId="...:time-less-than-or-equal"
    <Apply FunctionId="...:time-one-and-only">
    <EnvironmentAttributeSelector DataType="...#time"
      AttributeId="...:environment:current-time"/></Apply>
    <AttributeValue DataType="...#time">17:00:00</AttributeValue>
    </Apply>
  </Condition>
  </Rule>

  <Rule RuleId="FinalRule" Effect="Deny"/></Rule>
</Policy>
```

Figure 2.1: **XACML policy that only allows logins between 9AM and 5PM.**

as well as specifying authorizations for network resources and services [MBH99]. The IETF work concentrates on network services, like network classification.

Koch *et al.* [KKK96] define three levels of policy specification. The first, called *requirements* level, one declares the intended behavior of the policy in prose. The next step, done at the *goal–oriented* level, refines on the prose by defining the roles of the participating system objects in terms of constraints and actions. Finally, at the *operational* level, a policy description language (PDL) is used to express policy to be used by the runtime. We feel Koch's work is important because it identifies the need, and describes the manual steps, for a transition from a natural language to a machine understandable policy specification. Our vision is to improve on this process by automating this transition.

2.1.3 XML-Based Policy Languages

As we saw earlier trust management systems like TPL and Akenti leverage XML to define policy. The security assertion markup language (SAML) is another language that takes advantage of XML. SAML can define three types of assertions:

1. *Authentication assertions*, which provide information about how a subject has authenticated.

2. *Attribute assertions*, which associate subjects with their attributes.

3. *Authorization assertions*, which show whether a subject was permitted or not to access a resource.

SAML was designed to allow systems to exchange security information and not to specify security policy.

XACML, or extensible access control markup language, is another XML language designed to express security policy [GM03]. It is designed to complement SAML since it focuses on policy specification. XACML can be used to express standard authorization policies, namely whether a subject is allowed to perform an operation under a set of conditions. We feel that XACML also suffers from the complexity of XML syntax which is both verbose and low-level (see Figure 2.1).

2.1.4 Logic-Based Languages

In [JSS97], the authors present a formal logic language for specifying access control policies. It offers support for role-based access control. A disadvantage of formal logic-based approaches is that they are hard to map onto implementation mechanisms, and are not very intuitive due to the need for strong mathematical background. These limitations would make the task of administration even harder.

Minsky *et al.* introduce the notion of Law-Governed Interaction (LGI) [MU98, AMU01]. LGI is a message-exchange mechanism that allows a group of distributed agents to interact in a way explicitly specified by the security policy, also called the *law*. The law is defined over the various types of events (messages) that the agents can generate, and mandates the effects that such events should have. Enforcement of the law is performed by trusted entities called *controllers*. All controllers are uniform, and mediate the exchange of messages between the untrusted agents.

MulVAL [OGA05] is a security analyser that uses the Datalog model to analyse system vulnerabilities. The tool can be used to detect security policy violations that are caused by software bugs.

2.2 Policy Conflict Resolution

It is possible to define policy with positive statements, that allow operations and permit access to resources, and negative statements, that restrict permission. Furthermore, multiple users should be permitted to create policy. This can lead to potential conflicts when policy is evaluated. There have been a number of proposed solutions for policy resolution in case of conflicts [JSS97, LS97, CC97, LS99].

Systems that rely on *assertion monotonicity* like [KIGS01, BFIK99b]are able to achieve consistency since they rely exclusively on positive policy statements. All authority flows from a single top-level key, and it gets delegated to licensees, which can in turn further delegate privileges.

The Law-Governed Interaction [MU98, AMU01] approach relies on *controller* entities to mediate every access. These controllers are identical and they interpret the policy in the same way. This guarantees that all rules are interpreted the same way across the entire system.

Policy languages like Ponder [DDLS01] and SPL [RZFG01] assume the existence of a unique namespace for users, protocols, resources, *etc.* When rules operate on objects they are guaranteed to be referring to the same objects no matter where the rules are interpreted.

What we are interested in investigating is what happens when we cannot make such strong assumptions about common namespaces, identical controllers, and single-point policy evaluation.[1]

[1] As I will discuss in Section 4 we are interested in policies that involve several access points working together to correctly evaluate them.

2.3 Access Control Models

Traditionally access control models are split into *mandatory* (MAC) and *discretionary* (DAC). Mandatory access control policies are primarily interested with managing information exchange between objects of a systems. Discretionary access control policies are primarily interested with authorization of user access to information. An evolution of the discretionary access control model is *role based* access control (RBAC).

Mandatory access control policies, like the ones used in multi-level secure military systems, base their access control decisions on fixed rules, set by a centralized authority. Mandatory access control policies, as defined in the Bell-LaPadula [BL73] lattice-based model, address the issue of confidentiality. More specifically they focus on restricting information flow in a computer system. To accomplish this task, each subject (entities that execute actions) and object (entities that store information) is assigned a security classification. Based on this classification they form a lattice, which is then used to determine what actions subjects can execute on objects.

The Bell-LaPadula model, consists of a set of subjects, a set of objects, and an access control matrix. Additionally, the model has several ordered security levels. Each subject has a maximum clearance and each object has a classification which attaches it to a security level.

The set of access rights given to a subject are the following:

- **Read**: The subject can only read the object.
- **Append**: The subject can only write the object but it cannot read it.
- **Execute**: The subject can only execute the object but cannot read or write it.

- **Write**: The subject can both read and write the object.

The Bell-LaPadula model imposes the following restrictions:

- **Reading down**: A subject has only read access to objects whose security level is below the subject's current clearance level. This prevents a subject from getting access to information available in security levels higher than its current clearance level.

- **Writing up**: A subject has append access to objects whose security level is higher than its current clearance level. This prevents a subject from passing information to levels lower than its current clearance.

The Biba model [Bib77] modifies the Bell-LaPadula model by turning it "upside down" to provide data *integrity*.

2.4 Network Access Control Systems

Traditional firewall work [CB94, Mog89a, Mog89b, SSB+95, MWL95, GSSC96, NH98]. has focused on nodes and enforcement mechanisms rather than overall network protection and policy coordination.

In [HBM98], policy coordination is achieved with a role-based system where each principal may be issued a name by one service, on the condition that it has already been issued some specified name of another service. Event notification is used to revoke names when the issuing conditions are not satisfied, thus revoking access to services that depended on that name. Credentials are limited to verifying membership to a group or role, and

[HBM98] use delegation in a very limited way, obstructing decentralization.

Firmato's [BMNW99] "network grouping" language is locally customized to each managed firewall. The language is portable, but limited to packet filtering. It does not handle delegation or different, interacting application domains. Policy updates force complete reloads of the rule-sets at the affected enforcement points, and the entire relevant policy rule-set must be available at an enforcement point. This causes scaling problems as the number of users, peer nodes, and policy entries grows.

A similar system in [Hin99] covers additional configuration domains (such as QoS). Differences are the policy description language and the method by which the rule set is pruned for any particular device. Considerable work of this style has been done [Gut97, Mol95].

Another approach to policy coordination [HGPS99] proposes a ticket-based architecture using mediators to coordinate policy between different information enclaves. Policy relevant to an object is retrieved by a central repository by the controlling mediator. Mediators also map foreign principals to local entities, assign local proxies to act as trusted delegates of foreign principals, and perform other authorization-related duties. Coordination policy has to be explicitly defined by the security administrator of a system, and is separate from access policy.

In [BdVS00], the authors propose an algebra of security policies that allows combination of authorization policies specified in different languages and issued by different authorities. The main disadvantage of their approach is that it assumes that all policies and (more importantly) all necessary supporting information is available at a single decision point,

which is a difficult proposition even within the bounds of an operating system. Our observation here is that in fact the decision made by a policy engine can be cached and reused higher in the stack. Although the authors briefly discuss partial evaluation of composition policies, they do so only in the context of their generation and not on enforcement.

The NESTOR architecture [BKRY99] defines a framework for automated configuration of networks and their components. NESTOR uses a set of tools for managing a network topology database. It then translates high-level network configuration directives into device-specific commands through an adaptation layer. Policy constraints are described in a Java-like language and are enforced by dedicated manager processes, which pose scaling problems. We believe this approach has difficulty with decentralized administration and separation-of-duty concerns, due to its view of the network through a central configuration repository.

The Napoleon system [TOB98, TOP99] defines a layered group-based access control scheme that has similarities distributed firewall concept presented in [IKBS00], although it is mostly targeted to RMI environments like CORBA. Policies are compiled to access control lists appropriate for each application (in our case, that would be each end host) and pushed out to them at policy creation or update time.

RADIUS [RRSW97] and its proposed successor, DIAMETER [CRAG99], are similar in some ways to COPS [BCD+00]. They require communication with a policy server, which is supplied with all necessary information and makes a policy-based decision. Both protocols are oriented toward providing accounting, authentication, and authorization services for dial-up and roaming users.

A protocol for allowing controlled exposure of resources at the network layer is presented

in [Tsu92]. It uses the concept of "visas" for packets, which are similar to capabilities, and are verified at boundary controllers of the collaborating security domains.

Kerberos [SC98] defines a protocol for dynamically discovering, accessing, and processing security policy information. Hosts and networks belong to security domains, and policy servers are responsible for servicing these domains. The protocol used is similar in some ways to the DNS protocol. This protocol is serving as the basis of the IETF IP Security Policy Working Group.

The usefulness of decentralized access control management has been recognized for specific applications, *e.g.,* see [SP98] for an application of this approach in web access control.

[MNSS87] is an authentication system that uses a central server and a set of secret key protocols to authenticate clients and give both a client and an application server a secret key for use in protecting further communications. While enforcement is done in a decentralized manner, the system requires high availability and online security for the server. Furthermore, all network nodes are expected to have their clocks synchronized (typically within a few seconds of each other). The two most important deficiencies of Kerberos are that it does not implement any kind of authorization (applications are expected to make their own access control decisions, based on information they acquire through other means, *e.g.,* Directory Services, local ACLs, database queries), and it is expensive, in terms of administrative effort, to do cross-realm authentication, as this requires all clients to have complete knowledge of the trust relationships between realms (a Kerberos realm is the collection of systems and users managed by a single administrative entity).

The design principle of restricting local autonomy only where necessary for global

robustness has led to a scalable Internet. Unfortunately, this scalability and capacity for distributed control has not been achieved in the mechanisms for specifying and enforcing security policies.

The STRONGMAN system described in [KIGS01] demonstrates three new approaches to providing efficient local policy enforcement complying with global security policies. First is the use of a compliance checker to provide great local autonomy within the constraints of a global security policy. Second is a mechanism to compose policy rules into a coherent enforceable set, *e.g.,* at the boundaries of two locally autonomous application domains. Third is the "lazy instantiation" of policies to reduce the amount of state enforcement points need to maintain.

There are a number of deficiencies in the STRONGMAN architecture:

1. The architecture lacks support for cooperative policy specification and enforcement between different system nodes.

2. STRONGMAN defines interoperability at a very low level. STRONGMAN expects "higher-level" languages to produce KeyNote credentials that will form the policy to be enforced by the system. We feel that this makes the system harder to use in practice, than a system that uses a high-level policy specification since it does not address the issue of policy abstraction.

3. Additionally its monotonic architecture, that is, its lack of negative authorization, makes it unrealistic in some real-world scenarios.

2.5 Discussion

The existing security architectures that address the emerging models of distributed network applications suffer from poor support for administration. The components that comprise these systems are viewed as independent pieces and are managed as such, without the benefit of automation. Administrators have to configure a number of services, *e.g.,* web servers, databases, and compute farms, each with its own security requirements. Conventional mechanisms such as firewalls and compartmented file storage lead to *ad hoc* solutions that often prove inadequate for protecting heterogeneous distributed systems. This is due to their piecemeal configuration and lack of cooperation between components.

Policy based management relies on higher-level security policy languages to define system rules. While this is a step in the right direction, this type of policy system assumes that all nodes interpret policy the same way. Furthermore they assume that state is shared between all system nodes. In Chapter 3 we will be investigating what happens when system nodes don't all interpret the security policy in the same way, and what guarantees can we have that they all actually obey the same policy. Additionally, we will explore architectures where there is no single shared state between all system nodes, and how we can express and enforce security policy on such architectures.

Chapter 3

Security Policy Consistency

Our primary goal is to consistently enforce a single policy across a distributed system. One way in which we achieve this end is by applying the same high-level rules across all platforms. However, we need to interpret each rule by a per-platform adaptation layer.[1] How do we know that the adaptation is consistent across nodes?

The basic approach, outlined in Section 1.3.1, is to test whether the two-stage mapping of HLO to ILO, and ILO to LLO is consistent with the mapping M, of HLO directly to LLOs. However, these consistency checks are not free. They impose an additional burden on the user. In particular, M must be specified somehow, and before we can specify the mapping from HLO to LLOs, we first require a method of uniquely naming LLOs, and then a (partial) specification of M.

There are many ways of naming LLOs uniquely. For resources that have a "home" (*e.g.*, a file), we can concatenate the node ID of their home with the name of the ILO on

[1] A per-platform adaptation layer is the set of definitions that are used by the high-level language to map high-level objects to per-platform objects. For example, a definition that maps `TRUSTED_USER` to `root` on Unix, and `Administrator` on Windows.

that home. For others, (*e.g.*, a user), unique IDs may already exist (*e.g.*, user-id's). In cases not covered by the above strategies we can explicitly assign unique names.

3.1 Cross-platform Consistency

The basic method of testing for consistency is to see whether the two-stage mapping of HLO to ILO to LLOs is consistent with the mapping M. If M is not specified for a particular correspondence between an HLO and a set of LLOs, we simply check to make certain that the mapping on each platform is consistent with each other (see Section 3.1.2).

There are two points in time when we check for consistency: at "compile" time, when a mapping definition is installed on a given platform or a new high-level rule is specified, or at "runtime" when an access is ready to be resolved.

Compile time checks apply to a particular high-level rule. This rule has either just been added to the set of high-level rules in the system, or else a mapping definition that applies to this rule has been added or modified. We determine the set of all other high-level rules that may apply to a common object (recall that the HLOs in the high-level rules may refer to groups, or sets of objects). Using our algorithm for determining which rule applies, and walking over all the relevant rules, we can compute the set operations to express the subset of the HLO whose access is controlled by a particular rule. We partition the subsets into two classes, the subsets determined by rules whose decision is "allow", and those subsets determined by rules whose decision is "deny". We then take the union of all the subsets in each class. The system is consistent if we can show that these classes of objects are equal on all platforms. The system is inconsistent if we can find specific examples of LLOs that

have a different response on different platforms for a given operation. Section 3.4 gives a complete example of the above procedure. If set equality cannot be determined at compile time (possibly only for a specific set of objects), then the system cannot determine whether or not it is consistent.

The extent that consistency can, or cannot, be determined depends greatly on our knowledge of the set relationships between groups at the HLO, ILO, and LLO levels. Groups can be defined through finite enumeration. Such groups can be completely resolved. Groups can also be defined by set operations such as union, intersection, *etc.* If the groups are defined in high-level language through such operations, and the underlying H_p translations of the individual HLOs are either finite enumerations or can be shown to be equal across all platforms, then the resulting high-level groups can be completely resolved.

At both the policy language and adaptation layer, groups can be defined by escaping to external mechanisms. For example, one can imagine groups of files being defined by regular expressions over file names (`*.tex`, `*~`, *etc.*), or by Unix `find(1)`-like expressions (recent files *i.e.,* `atime -2`), or even by arbitrary predicates ("PREDICATE?(file)"), where PREDICATE? can check whether a file satisfies PREDICATE? by any means it needs to, even looking for file properties inside the file system. The only requirement on these external mechanisms are that they implement (partially maybe) MEMBER-OF, SUBSET?, SUPERSET?, DISJOINT, that can answer TRUE, FALSE or INDETERMINATE. If the policy language uses such an external mechanism, the adaptation layer must map it to an external mechanism of its own that preserves the set relationships at the policy language level, and also implements the set functions required above. If an escape mechanism is meaningful at the policy language level, though, (meaning that it is platform independent), then it is

almost certainly meaningful at the adaptation layer level, too, and the adaptation layer implementation is the escape mechanism itself. However, we expect the most typical use of such escape mechanisms will be only in the adaptation layer definition; a policy language HLO would map to an ILO group defined by one of these escape mechanisms.

Finally, a policy language HLO may map to an ILO that is itself a primitive group. Examples of such groups can be Buddy-Lists for AIM [aim], or `groups` in the Unix sense [RT74].

Often the reason that consistency cannot be guaranteed is that the group or object is not yet instantiated at the time of rule compilation. For example, the results of a query cannot be instantiated statically, because its value only becomes known at runtime. These rules can be flagged to mark the possibility of an inconsistency, and then they can then be checked against the security policy at runtime. In cases where an inconsistency cannot be resolved at runtime, we deny access and report it to the user.

The most basic check for consistency is that a query about whether a given agent can perform a particular operation on a given LLO must get the same response on any platform. Assuming we have some way of posting this query without performing the operation,[2] then we can perform this comparison regardless of whether the system has separable security mechanisms, or whether M is specified or not. It is even possible to take a pro-active approach, and actively and continuously test the system for policy inconsistencies. For example, we could periodically look at randomly chosen objects within groups, and make sure they behave the same on all platforms. The objects may not even be

[2]Another alternative is to record the result of such a query when the operation actually occurs, caching the result, and comparing the result when/if the same operation is performed on some other platform.

chosen "randomly", but select them using technology from test-vector generation and/or the model-checking communities, to find particularly useful examples to check. I suggest these approaches, but have not yet explored them in any depth.

3.1.1 Current Approaches

Before we continue, we will briefly remind the reader how some related systems address policy consistency. Conflict resolution in security policy is one aspect of policy consistency. As we have already discussed in Sections 1.2 and 2.2, there has been extensive work in the literature on how to resolve possible conflicts.

Systems that rely on *assertion monotonicity* like [KIGS01, BFIK99b] are able to achieve consistency because they rely exclusively on positive policy statements. All authority flows from a single top-level key, and it gets delegated to licensees, which can in turn further delegate privileges. Monotonicity offers a "clean" way of expressing security policy rules. Unfortunately, it is an unnatural way since humans often express rules as negative assertions and exceptions. This can make the above systems very clumsy to use, and also verbose (where positive rules have to express the desired policy). Furthermore, this model does not allow lower level administrators to refine policies of higher ups.

The Law-Governed Interaction (LGI) [MU98, AMU01] approach relies on *controller* entities to mediate every access. These controllers are identical and they interpret the policy in the same way. This guarantees that all rules are interpreted the same way across the entire system. The LGI approach is similar to the Distributed Firewall and STRONGMAN [IKBS00, Ker01], which also rely on a single common language and multiple similar access control points. While this model is attractive due to its simplicity, it has

the drawback that it requires a homogeneous architecture.

Finally, high-level policy languages like Ponder [DDLS01] and SPL [RZFG01] assume the existence of a unique namespace for users, protocols, resources, *etc.* When rules operate on objects they are guaranteed to be referring to the same objects no matter where the rules are interpreted. Even though rules operate on the same objects, it is possible for objects to map on different entities on different platforms. There is an implicit assumption that all platforms have the same way of naming same object.

3.1.2 Consistency Procedure

In the previous section we gave a largely intuitive description of how we achieve policy consistency on a set of heterogeneous platforms. We will now describe in more detail the actual consistency procedure. Let us start by identifying the cases during which a consistency check is necessary:

1. **At system initialization.**
2. **When adding or modifying a policy rule.**
3. **When adding a new platform.**
4. **When adding or modifying a policy to mechanism mapping.**
5. **When adding or modifying a mechanism to reality mapping.**

We represent policy rules as *3-tuples* $P_i :< S_i, A_i, T_i >$ that represent whether a subject is permitted to perform an action on a target.

- *Subjects* – objects which perform actions as specified by policy.

- *Actions* – the possible actions subjects can perform.

- *Targets* – objects that the subjects perform the actions on.

S_i, A_i, and T_i represent objects, or sets of objects, in the high-level policy language. These objects, as we already discussed in Sections 1.3.1 and 3.1, are mapped (H_p) onto mechanism-specific objects (or sets of objects) that have local meaning to each platform. Finally, the mechanism-specific objects are mapped (L_p) onto physical-reality objects (or sets of objects) that represent unique users, resources, *etc.*

Given an input of policy rules, along with the corresponding hierarchy of mappings our procedure (see Figure 3.1) is able to determine whether the policy is consistent or not. Furthermore, in cases of inconsistency it reports the violating rules, the platforms where the conflicts occurred, and the offending objects.

Figure 3.1 presents the pseudocode of our security policy consistency algorithm. The algorithm is composed of two parts. In the first part it generates all possible policy rule states. In the second part it checks for policy inconsistencies. Since this is an exhaustive algorithm, if all rules, mappings, and objects are defined statically, it is guaranteed to return all possible inconsistencies.

To generate all possible policy rule states, it uses its input of high-level policy rules, represented by 3-tuples, along with the HLO-ILO and ILO-LLO mappings. Consequently it uses the mapping to expand the every rule for every platform in the system. After it has generated the rule state space for every platform, it proceeds on an exhaustive pairwise comparison of every state. The comparison produces all possible rules inconsistencies.

```
Input: High-level rules in tuple form, mappings H_p and L_p
Output: Types of conflicts, conflicting rules and objects


/* First expand the high-level rules for every platform */
For every platform p
        For every policy rule r
                For every High-level mapping H_p
                        For every Low-level mapping L_p
                                Expand policy rule r
                                Store expanded policy rule r
Return expanded policy rules


/* Given the expanded rules try to find conflicts across platforms */
For every pair platform p_i platform p_j
        For every pair of expanded policy rule r from platform p_i and platform p_j
                Compare rules
Return types of inconsistencies, conflicting rules and objects, and platforms where the
inconsistencies were detected
```

Figure 3.1: **Pseudocode of the core consistency procedure.**

The running time of the first part of the algorithm is $O(|P| \times |R| \times |H_p| \times |L_p|)$. As we can see it is highly dependent on the *richness* of the mappings between object layers. In the worst case every HLO will be mapped to every ILO, and every ILO will be mapped to every LLO, even though such a case would not make any sense. Nevertheless, in such a case traversing all the mappings would require, $O(|H_{LO}| \times |I_{LO}| \times |L_{LO}|)$, where $|H_{LO}|$, $|I_{LO}|$, and $|L_{LO}|$ represent the number of HLO, ILO, and LLO objects in the system. This will give us a total worst case running time of $O(|P| \times |R| \times |H_{LO}| \times |I_{LO}| \times |L_{LO}|)$ for the first part of the algorithm.

Assuming a worst case scenario, after expanding the policy rules we will have a complete set of all 3-tuples of LLO of order $O(|L_{LO}|^3)$, since the rules were expanded all the way

down to the unique physical-reality objects.[3]

The running time of the second part of our algorithm is therefore $O(|P|^2 \times |L_{LO}|^6)$, due to the pairwise comparison of platforms and rules.

This yields a total running time of $O(|P| \times |R| \times |H_{LO}| \times |I_{LO}| \times |L_{LO}| + |P|^2 \times |L_{LO}|^6)$. In reality we expect running times to be significantly lower, since it is unrealistic and unlikely that all possible mappings will be defined. Furthermore, this is a cost paid *only* at system initialisation, as changes occur in the system, consistency checks are *only* limited to the rules, mappings, or platforms that are modified.

Adding or modifying rules

When adding a new rule, we only need to compute the newly created expanded rule states. Also at the comparison phase, we only need to compare the newly created mappings. This gives us a running time of $O(|P| \times |H_{LO}| \times |I_{LO}| \times |L_{LO}| + |P|^2 \times |L_{LO_{new}}|^2)$. We can treat modifications as a rule removal followed by an addition.

Adding a new platform

When adding a new platform, we only need compute the newly created expanded rule sets for the platform. We also avoid the complete pairwise comparison between platforms. This gives us a running time of $O(|P| \times |H_{LO}| \times |I_{LO}| \times |L_{LO}| + |P| \times |L_{LO}|^6)$.

[3]Keep in mind that 3-tuples are composed of subjects, actions, and targets. Subjects and targets can have unique "physical-reality" instantiations. For actions we assume that in "physical-reality" there are unique types of operations, like reading, writing, *etc.*

Adding or modifying mappings

Finally, in the case where object mapping change, we only have to create expanded rule states for the rules that are affected by those changes and not for the complete rule base. Complexity in this case becomes $O(|P| \times |R|' \times |H_{LO}| \times |I_{LO}| \times |L_{LO}| + |P|^2 \times |L_{LO}|^6)$. Again we can treat these modifications as a rule removal followed by an addition.

Dynamic Objects

The discussion so far assumed that all objects are statically defined. There are cases however where we cannot make such an assumption. For example, there are cases where objects are created dynamically as a result of queries, special predicates, on-the-fly group creation, *etc.*, as we discussed in Section 3.1.

More specifically we can have high-level policy object definitions but their lower-level representations might not yet exist. In such instances, our consistency procedure defers the consistency check until they are instantiated at runtime. The runtime system will then invoke the consistency procedure, and using the newly instantiated objects, it can detect whether the policy will remain consistent.

3.2 Preservation of Intention

Our architecture primarily focuses on consistent enforcement of the high-level policy rules across all platforms. But although, as we already argued in Section 1.3.1, there is very little we can do to assure preservation of intent, there are tools we can provide that make it easier for policy specifiers to understand the implications of their rules.

The main tool we provide is the *policy debugger*, a tool that allows a user to query the system about rules and authorities applied to specific objects in the system. This allows troubleshooting of inconsistent or otherwise problematic policies and helps ensure that the specification matches the intent of the writer. It can also aid administrators to determine what mappings must be implemented when a new platform is added to the system.

The first function of the policy debugger is to mimic the behavior of our policy system and find all rules relevant to a particular object or agent.

From first principles it would seem advantageous to keep all security policy rules in a central repository so that one could inspect all the rules to understand the "entire policy." A moment's reflection reveals that such an undertaking is undesirable. Consider again the example of a file system. In effect, there are many rules (either explicit or implicit) controlling access to *each* file in the system. Anyone attempting to acquire a complete picture of the corporate policy by inspection of the complete pile of per-file rules would be lost in a sea of detail. Specifically directed queries are likely to be more valuable. For example "What rules apply to members of group X performing operations Y on the class of objects Z?", and which of those rules specifically determines the outcome in a particular situation? Alternatively, "Who has the authority to grant access to allow X to perform Y on Z?"

The policy debugger takes high-level policy rules with wildcards and returns all rules that apply. For example, if a rule is passed to the policy debugger with the Target and Subject fixed to specific HLOs, and the Action as *, then the policy debugger will return the rules that apply to that Agent operating on that resource for *all* operations.

Used in conjunction with our security policy consistency procedure, security officers

can gain greater confidence that their computing system operates as intended.

3.3 Handling inconsistencies

If an inconsistency is detected then the user must be informed, and two decisions must be made; one by the system and one by the user.

The system must decide how to resolve the inconsistency temporarily; the user must decide how to permanently resolve the inconsistency. The system has three choices: (*i*) it can deny the operation on the platforms on which it is currently allowed, or (*ii*) it can allow the operation on the platforms on which it is currently denied, or (*iii*) it can allow the inconsistency to stand until the user resolves it. It seems preferable to us to let the inconsistency stand and wait until the user resolves it. It seems insecure to allow access on all platforms, and it seems dangerous to deny it on all platforms (one may disallow the access needed to resolve the inconsistency!). More experience may alter this analysis.

The inconsistency may be resolvable by the user in one of two ways, corresponding to whether the inconsistency was introduced by an error in the HLO-ILO mapping, or by an ambiguity in the high-level rule. In the first case, the inconsistency may be resolved by simply modifying a relevant mapping definition on a particular platform to be consistent with others. In the second case, additional high-level rules may need to be introduced to further discriminate between members of a particular set, or, rules may need to be removed for the same reason.

3.4 Example

To illustrate how policy inconsistencies can affect the security properties of a system, we present a real world policy consistency failure in one of the file servers of our department. We use a file system example since most people are familiar with file access control and it represents a clear easily understood failure. Our system, to be described in detail in Chapter 5, however is not limited to just these simple cases.

Our file system server holds the home directories of 406 users and provides mail services. Furthermore it exports file systems to a number of clients. Among the exported file systems are the e-mail directories, exported as read/write to 3 hosts, and all the home directories, exported as read/write to 17 hosts and read-only to 3 hosts, to let the owners of the clients hosts have access to their home directories and e-mail.

This scheme unfortunately causes a failure in the security policy, which requires users to *only* have access to their *own* files, since the owners of the client machines can gain control of other users files.[4] It is tempting to brush away the breakdown in security policy as a shortcoming in the configuration, as well as the authentication and authorisation model of NFS, however the problem is deeper than that.

The problem arises since we have multiple platforms that identify objects (in this case users and files) in different ways. This leads to an inconsistency in the way the same rule (in this case that each user should only have access to their own files) is applied at the local file system and at the remote file system. It is easy to imagine the critical nature of such breakdowns once we consider how extensive the use of network file systems such as

[4]The owner of the client machine can create duplicate accounts with the same user id's as on the server machine and subvert the security policy.

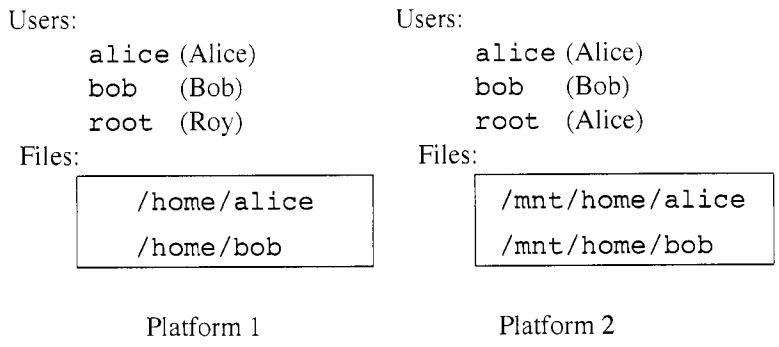


Figure 3.2: **Remotely mounted file systems.**

NFS and SMB are in our modern computing environments.

Using the process described in Section 3.1 we are capable of statically catching this type of inconsistency, in cases where the sets of objects in the system are finite enumerations. To understand the above failure let us walk through a more simplified scenario. We assume two hosts, *Platform 1* and *Platform 2*, with three users, Alice, Bob, and Roy, and their corresponding files (Figure 3.2). Roy is superuser on *Platform 1* and Alice is superuser on *Platform 2*. Furthermore *Platform 1* is exporting the user files to *Platform 2*. Our security policy specifies that each user should *only* have access to their *own* files.

We first define the H_1 and H_2 mappings for each platform that will map the high-level policy language objects (`USER_ALICE, USER_BOB, USER_ROOT, ALICE_FILES, BOB_FILES, ROOT_FILES`) to the platform-specific objects (`alice, bob, root, /home/alice, /home/bob, /mnt/home/alice, /mnt/home/bob`). Then we define the L_1 and L_2 mappings of the platform specific-objects to the physical-world objects (see Figure 3.3). We are independent of the type of high-level language used and to the method of naming objects uniquely. Using our algorithm from Section 3.1 our compiler composes the mappings for each platform and compares them to each other. A conflict appears due to the fact that the two platforms

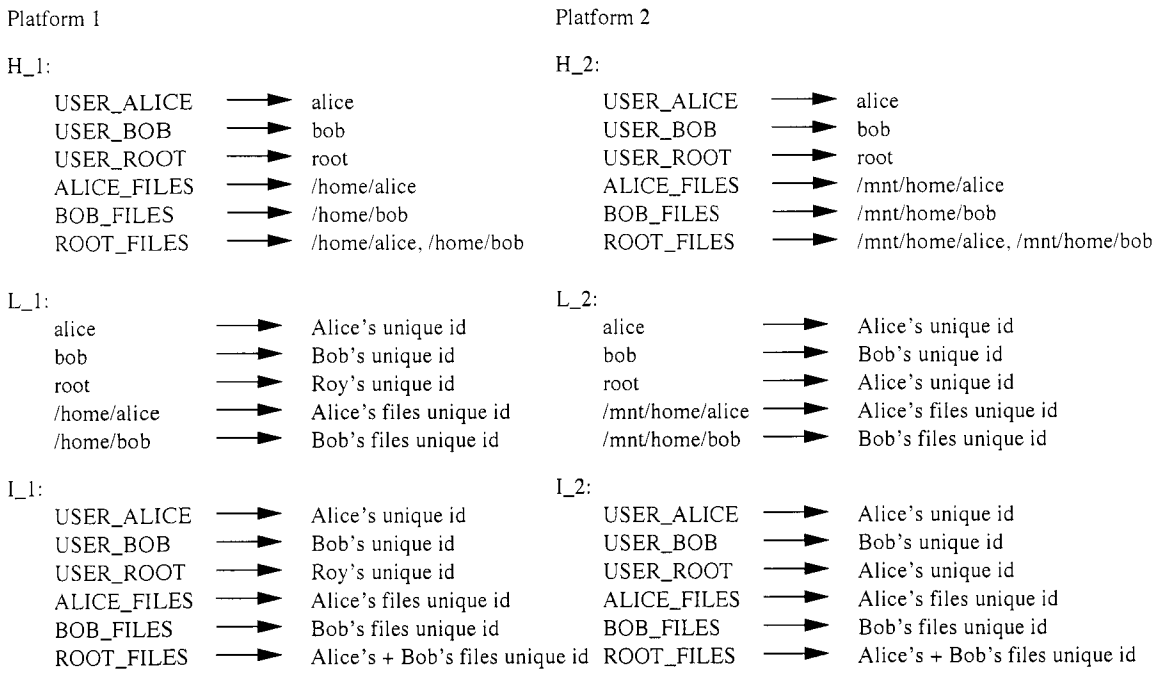


Figure 3.3: **Mapping of HLO to ILO and ILO to LLO, along with their composition, for two different hosts.**

have a different specification for the superuser (see Figure 3.3).

In more dynamic cases, where sets of objects are unknown or dynamically created, and the compiler cannot statically determine a conflict, we can rely on our runtime system to either catch them, deny access, and report to the user.

3.5 Discussion

In this Chapter we addressed the question of policy consistency in computer environments where we cannot make such strong assumptions about common namespaces, identical controllers, and single-point policy evaluation.

Our definition of consistency is not the one traditionally used by security policy systems, that is, a security policy that is free of conflicts. Instead we defined two other notions,

one of cross-platform consistency (see Section 3.1), and one for preservation of intent (see Section 3.2).

In this context we proposed a framework for representing how high-level policy objects are mapped to heterogeneous mechanism-specific objects, and then consequently to physical-reality objects. Given such a mapping we described and analyzed a novel procedure for determining security policy consistency in heterogeneous decentralized systems. The procedure is capable of statically determining consistency in those cases where objects are known at policy specification time. In the event that objects are unknown or undefined at specification time, our procedure reports the chance of an inconsistency, and defers the consistency check to the runtime environment.

Finally, we demonstrated how our procedure works in the small and simple case of only a few objects. This example was drawn from our experience with real-life systems. The policy failure even in such a simplified scenario, supports the case for an automatic policy consistency procedure.

Chapter 4

Security Policy Cooperation

Security is an application-dependent property, with some applications requiring very little "support," while others require considerable infrastructure to support their privacy, integrity and availability requirements. When applications were confined to a single computer, application programmers could rely on the host operating system to support these requirements. The advent of the Internet has introduced new challenges for applications with non-trivial access-control requirements. In particular, various network access-control mechanisms such as firewalls are largely oblivious to applications (and vice versa), while file-access privileges associated solely with users may not allow for sufficiently fine-grained access control to handle safety issues related to untrusted active content (such as JavaScript applets). In this section, we introduce the notion of *cooperative policy evaluation*, which captures in a policy specification the complete access-control requirements of a service. This single policy specification can then be used by enforcement mechanisms in hosts, routers, firewalls, and elsewhere to produce a consistent environment for the service.

To illustrate the need for cooperative policy evaluation, let us look at an example. Consider web services run on a virtual web server consisting of tens or hundreds of machines in a server "farm," with co-located auxiliary services such as a database, credit card transaction support, and web-mail service. Figure 4.1 shows the components of such a system, without elaborating on the replicas of each component used in a full-scale implementation (*e.g.*, multiple servers per physical location, and replicated physical locations, each with a database replica and a credit card support system).

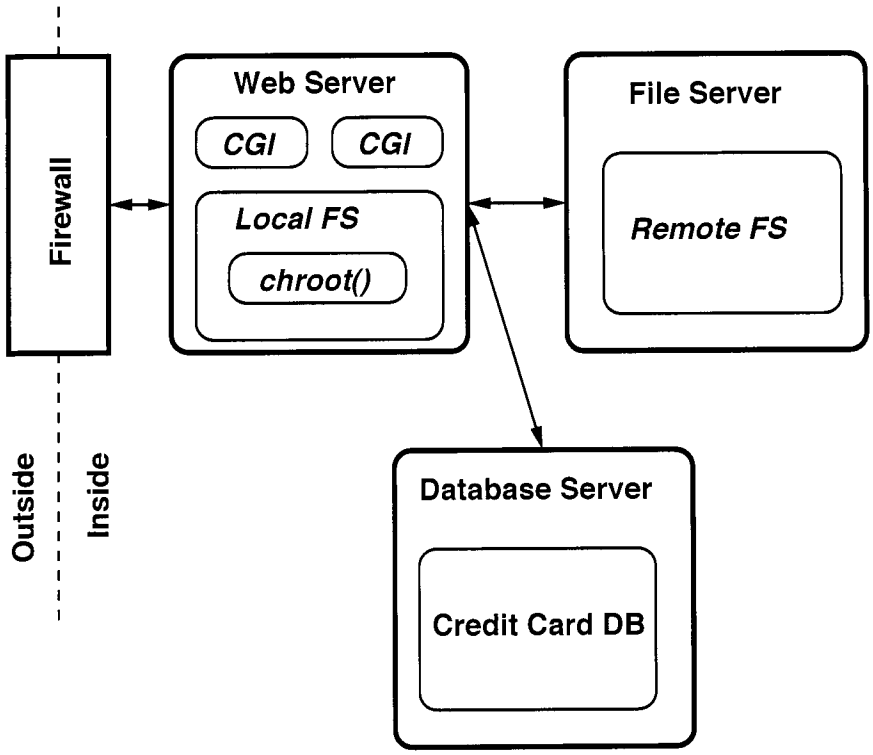


Figure 4.1: **Current generic architecture of distributed application clusters.**

Typically, the configuration of such a system is static. By this, we mean that the administrator configures each component independently, and depends on the correctness of the individual policies to enforce a system-wide policy for any particular user or class of

users. There is no coordination among the nodes in the system, nor is there any coherent relationship between the network access control (achieved with firewalls and routers) and the node access control (achieved with compartmented file systems and process control). The application components thus become very difficult to manage effectively, and misconfiguration and other administrative errors occur [How97, Woo01]. The causes can range from the difficulty of managing local components, the difficulty of managing local scale, or the difficulty of coordinating across sites and administrators. Such systems call for an approach that can ensure consistent access-control policies, as well as meet application-specific requirements using shared resources such as hosts and the network.

Cooperative policy evaluation requires coordination among clients, servers, and networks to deliver a reliable, secure service to clients. It permits security policy to encapsulate the security requirements of all participating parties. That is, it enables multiple system elements to work together towards enforcing more complex, distributed policies.

It is important at this point to collate the notion of *consistency*, which we introduced in the previous section, with the notion of *cooperation*, which we will be discussing in this one. Policy consistency addressed the question of determining how security policy defined for heterogeneous platforms, each with its own platform-specific abstraction, complies with the global security policy. Policy cooperation addresses the question of how security policy can be defined and evaluated on decentralized nodes, with multiple nodes participating in the evaluation of policy statements.

4.1 Security Policy in Decentralized Systems

Large distributed systems cannot be administered one machine at a time. Many tools, like those discussed in [Koe84, RjL88, Smi89], as well as others, have been built to ease the task of administering multiple computers. However, for the most part, these tools have been concerned with *configuration file* management and synchronization/distribution, rather than policies. Policy configuration files can be centrally administered, but this is more a side-effect than a basic premise of the distribution tools. The problem is that, today, complex policies must be expressed in many different ways. This is due to the fact that modern computer systems have grown in scale and complexity (see Section 1.1). For example, consider again the network shown in Figure 4.1. Assuming there is a security policy *"barring improper access to the credit card database,"* the question is how to best architect a system that can enforce this policy.

The first obvious step is a firewall rule that blocks access from the outside. However, we also need to guard against attacks originating from the firewall-protected part of the network, either by insiders or from inside machines that have been compromised. Accordingly, the credit card database may have its own configuration and policy rules blocking most access from "inside" machines. Indeed, it may be protected by its own packet filter or firewall. However, not all users that can legitimately access those "inside" machines should necessarily be trusted. Accordingly, additional access rules may be needed as well. These may be lists of cryptographic credentials to be accepted, or they may be distributed firewall rules [IKBS00, Bel99], *etc.* For that matter, the database system may itself have access control mechanisms that need to be configured.

It is clearly impractical to try to configure each of these four systems separately. While current tools can easily distribute policy files, the deeper problem — ensuring consistent access policies across many different systems — is far more difficult. It is this problem that we are addressing. Our task is further complicated when enforcement must be split across different components. For example, a rule that says *"user A may access database column B on server C when coming from machine D via IPsec"* should be specified in one place. Enforcement, however, could be split between a firewall that permits access to the database port from D and a firewall rule on D that recognizes A's credentials, while enforcement of access restrictions to particular fields must be done in the database server itself.

One attempt to solve this problem in a limited domain is the Firmato [BMNW99] firewall language. Firmato is a high-level language for specifying firewall policies. The administrator specifies a policy and a network topology; the policy is then compiled into rule-sets for the different firewalls (which may be from different vendors), and distributed to each firewall protecting the domain described in the topology. While this is certainly a step in the right direction — a single policy statement can simultaneously control several different firewalls — it is limited to a single application *class*. Firmato is just an example of a domain specific language, created for the purpose of specifying firewall policies in a network, among many such specialized languages designed to control other system components such as the file system, processes, and others ([AR00, BBS95, GWTB96, WSB+96]). As noted above, complex — *i.e.,* realistic — security policies need to simultaneously control many different *types* of applications.

We can thus list our requirements for an effective, multi-element security mechanism:

1. The input language must be rich and extensible, in order to be able to express a wide variety of policies, for a wide variety of devices and applications.

2. The input language must be high-level, to avoid unnecessary device-specific semantics.

3. There must be a reliable compilation and distribution mechanism that will distribute the policy to all relevant system nodes.

4. The policy must be enforceable by trusted components alone.

5. Nodes *must* be able to exchange policy decisions as well as state.

We are not the first to propose some of these requirements ([IBS01, IKBS00, IBI+03]), similar recommendations have also been made by others (*e.g.,* [Dam02, TJM+99, DLSD01, Mar97], *etc.*). The new addition we bring to the list of requirements is the need for cooperation between security nodes. Before we expand on our proposed notion of cooperation, let us first briefly look at the system requirements for an effective security policy model in decentralized systems.

4.1.1 Requirements

Input Language

Most organization use a variety of mechanisms to control different systems and data. Each of those mechanisms usually has its own security policy specification method. The collection of all the policies defined by each of these mechanisms comprise the global security policy for that organization [RZFG01, DBSL02]. Due to the heterogeneous nature

of such systems, security policies are scattered in multiple places using different definitions that make them hard to administer and understand.

This obvious shortcoming has been identified, and in the last few years there have been several proposals for security policy languages, such as [GM03, HMM+00, TJM+99, BFIK99b] and others, designed to replace the plethora of device-specific security policy definition methods.

Abstraction

As we mentioned in the previous section there is a large degree of heterogeneity in most organizations. To address the challenge of specifying security policy for such systems researchers have proposed abstracting individual configurations by using higher-level policy languages ([DDLS01, KKK96, RZFG01]).

The advantages of such an approach is that different systems can now be configured using a single high-level interface. Of course this assumes that we can modify the back end systems (*e.g.,* file systems, operating systems, firewalls, *etc.*) accordingly, to make them capable of interpreting the newly introduced policy specification language ([MPI+03, IBI+03, IKBS00, Dam02]).

Distribution

Since we are focusing on decentralized systems an important question is how to accomplish the distribution of policy to all participating nodes. There is no shortage of solutions in this area (*e.g.,* [DLSD01, TJM+99, IKBS00, Ker01] *etc.*). Some systems use centralized policy repositories from which enforcement points pull the policy, others rely on a more

pro-active push model, while others might employ multiple policy generation points, *etc.*

As for policy revocation, there exist various revocation methods (Certificate Revocation Lists (CRLs), Delta-CRLs, Online Certificate Status Protocol (OCSP), refresher certificates), each with its own tradeoffs in terms of the amount of data that needs to be maintained and transmitted, any online availability requirements, and the window of vulnerability.

Enforceability

The policy must be completely enforceable by trusted components alone (dedicated nodes, operating system kernels, *etc.*), without the cooperation of user-level processes on marginally-trusted machines. However, the definition of a trusted component should be extensible such that a finer-grain policy could be enforced under certain assumptions (*e.g.,* a database that enforces access restrictions to specific columns).

Ideally the trusted components will be able to use a single policy language that abstracts mechanism specific details. In cases where such integration is not possible the security policy should be mapped to the appropriate format for the applications it is intended to control [Dam02].

Cooperation

Security policy in decentralized systems is defined for each system element and it completely governs that elements behavior. As we saw, policy can be defined, modified, and enforced in a variety of ways depending on the underlying system requirements. However, existing systems, for the most part, are interested in capturing and expressing the security

requirements of each individual system node, without considering interaction with other nodes. For example file system access control works independently from firewall access control.

As we began discussing in the beginning of this chapter, there are situations where an inter-node cooperation scheme that allows for multiple nodes to work together towards the evaluation and enforcement of the global security policy would be beneficial. In the remainder of this chapter we will investigate such a cooperative scheme and look at the advantages it offers over traditional approaches.

4.2 Cooperative Policy Evaluation

Security policy rules apply to system resources and each enforcement element is obliged to carry out the specified policy. We have seen that rules can be centrally defined and managed (*e.g.,* [Dye88, RjL88, SL93, LSDD00], *etc.*), defined in a distributed fashion (*e.g.,* [MPI+03, LPI+03, KIGS03a], *etc.*), or even defined per application instance in an *ad hoc* manner, such as packet filtering rules in firewalls and access control permissions in file systems. Each of these approaches offers its own advantages and disadvantages on security policy specification, management and enforcement.

Regardless of how policy is specified it is important to note that each access point is responsible for enforcing the security policy that applies to it, and it does so largely independently of other access points.

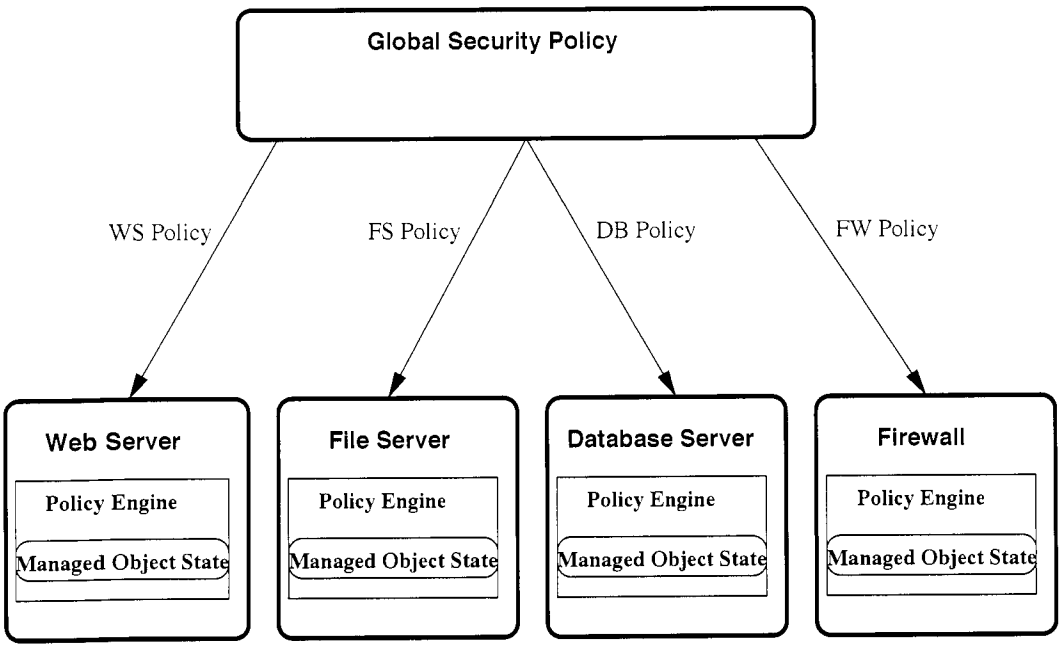


Figure 4.2: **Today enforcement nodes don't cooperate when enforcing the system-wide security policy. Instead, each node is responsible for enforcing its private security policy. Figures 4.3 and 4.6 present how the model can be extended to permit cooperation between enforcement nodes.**

4.2.1 No Cooperation

The most typical scenario in organizations today is for each application to be operating independently. Regardless of how policy is specified (high-level vs. low-level), managed (distributed vs. centralized), *etc.*, each application has its own policy definition and works independently towards enforcing it (Figure 4.2).

This approach offers a straightforward and simplified way to think about security policy since each application is addressed individually. It shields the administrator from having to address dependencies between applications and also eliminates any possible unexpected side effects caused by application interaction. We can see the value of these properties in certain kinds of environments, but at the same time we feel that such constraints limit the

potential of more powerful and expressive security policies.

If we recall our example policy rule (*"user A may access database column B on server C when coming from machine D via IPsec"*) from Section 4.1, it becomes immediately apparent how the per-application definition of security policy is handicapped. The only way for the above policy to be implemented correctly, and ensure that user A can actually access column B on server C, is to force the firewall to drop connection attempts from user A unless specifically coming from machine D via IPsec.[1]

4.2.2 Limited Cooperation

It is clear from the previous discussion, that security policy systems often need to share some state about the objects they control. An example of such a system is Ponder [DLSD01, YS96], in which a Domain Service (using an LDAP server) holds reference to all managed objects (see Figure 4.3).

With this approach the policy system uses a single managed object space. Enforcement points can uniquely reference managed objects permitting a straightforward way to express policy rules. This simplification however has the drawback that it assumes that every node will have the same object representation, and objects exist uniquely inside the object hierarchy.[2] In decentralized heterogeneous systems this is a limiting assumption.

In the Distributed Firewall [IKBS00] it became apparent that to effectively make policy decisions, the policy manager had to extract information from lower layers of the network protocol stack (TCP, UDP, IPsec, IP, *etc.*). That is, there was a need for some primitive

[1]Provided that the firewall can actually determine the identity of the user trying to connect.

[2]It also introduces a single point bottleneck, but bottleneck investigation is outside the scope of this disseration.

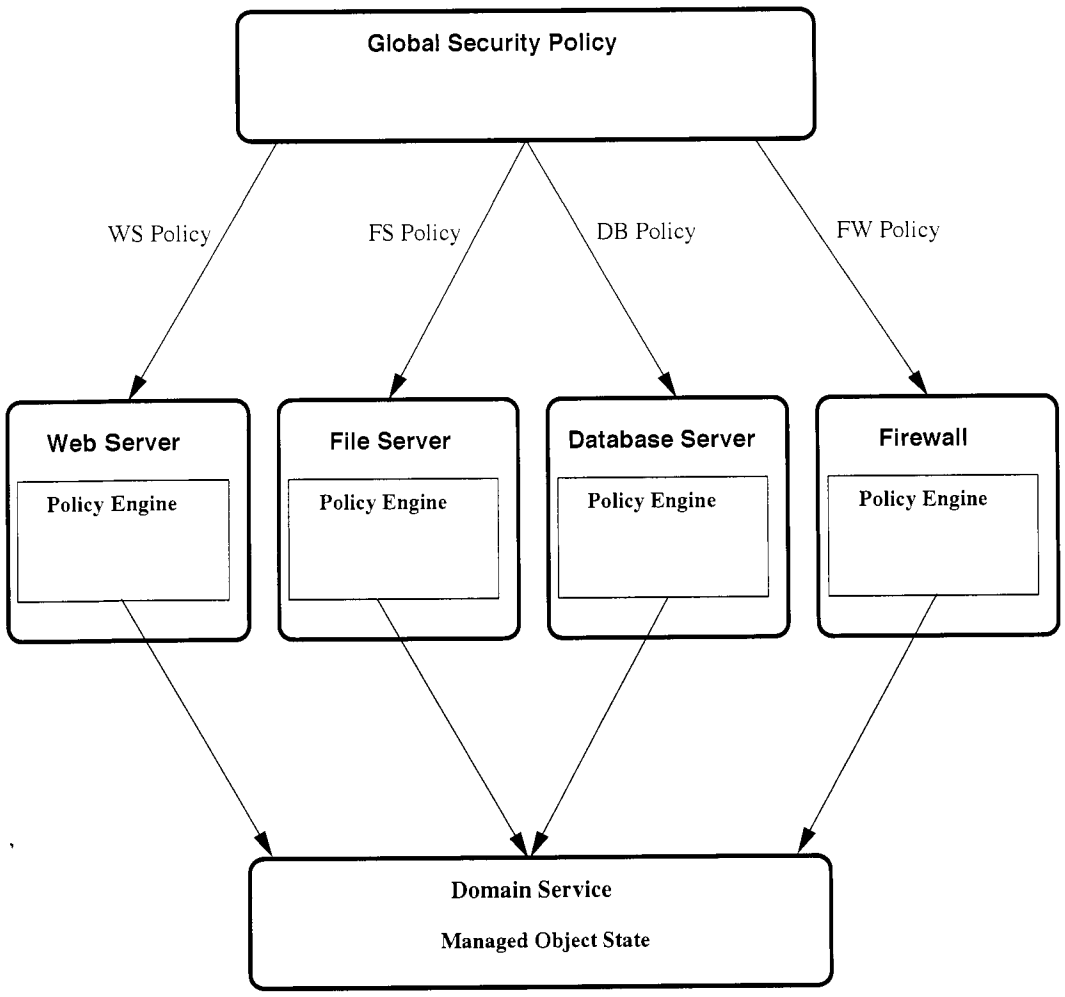


Figure 4.3: **Nodes enforce their specified security policy but they share policy object space using a domain service.**

form of cooperation between the policy manager and the network layer.

Later systems, such as STRONGMAN [Ker01, KIGS03b] and VPS [IBS01, IBI+03], used a similar method, called *referrals*, for communicating information across enforcement layers. In the examples in Figures 4.4 and 4.5, the Apache web server requires the use of strong encryption to serve some documents to user FOO. The IPsec layer will test whether user FOO has all the prerequisites to indicate that he is using strong encryption.

```
Authorizer: ADMINISTRATOR_PUBLIC_KEY
Licensees: FOOS_PUBLIC_KEY
Conditions:
        app_domain == ''IPsec policy'' &&
        esp_present == ''yes'' &&
        ( esp_enc_alg == ''3des'' ||
          esp_enc_alg == ''AES'' ||
          esp_enc_alg == ''blowfish'' ) &&
        ( esp_auth_alg == ''hmac-sha1'' ||
          esp_auth_alg == ''hmac-md5'' ) &&
        ( local_filter_type == ''IPv4 subnet'' ||
        ( local_filter_type == ''IPv4 range'' ||
        ( local_filter_type == ''IPv4 address'' ) &&
        local_filter_addr_lower >= ''192.168.001.000'' &&
        local_filter_addr_upper <= ''192.168.001.255''
                -> ''strong encryption'';
Signature: PUBLIC_KEY_SIGNATURE_FROM_ADMINISTRATOR
```

Figure 4.4: The IPsec layer associates a value of strong encryption whenever it is configured a certain way. Cryptographic keys have been replaced with symbolic names, in the interest of readability.

```
Authorizer: WEB_ADMINISTRATOR_PUBLIC_KEY
Licensees: FOOS_PUBLIC_KEY
Conditions:
        app_domain == ''apache'' &&
        method == ''GET'' &&
        uri ~= ''SOME_URI'' &&
        source_address_lower >= ''192.168.001.000'' &&
        source_address_upper <= ''192.168.001.255'' &&
        ipsec_result == ''strong encryption'' &&
                -> ''true'';
SIGNATURE: PUBLIC_KEY_SIGNATURE_FROM_WEB_ADMINISTRATOR
```

Figure 4.5: Rule that permits access to the web-server provided that the result we extracted from the IPsec layer indicates the use of strong encryption. Cryptographic keys have been replaced with symbolic names, in the interest of readability.

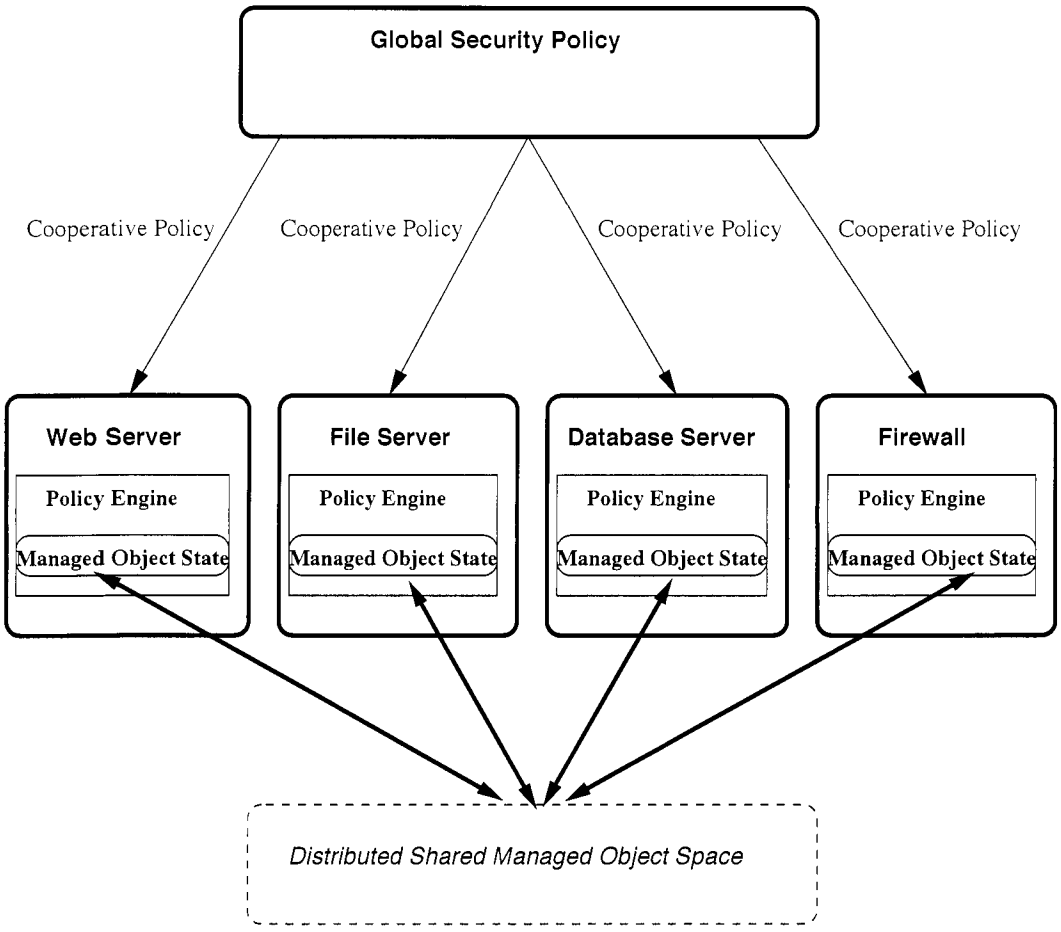


Figure 4.6: **Nodes enforce the security policy in a cooperative fashion. The nodes can exchange information using remote queries to other nodes, creating a distributed shared space for the state of the managed objects.**

4.2.3 Complete Cooperation

As we defined in the beginning of this chapter, policy cooperation addresses the question of how security policy can be defined and evaluated on decentralized nodes, with multiple nodes participating in the evaluation of policy statements. The idea is that security policy rules are composed of smaller pieces, and each piece can be evaluated on a different enforcement point. At the end of the evaluation, results are returned to the node that initiated the policy evaluation, where they are composed and a decision is reached.

Such a structure gives us the ability to write policies that span the usual boundaries of applications, hosts, operating systems, *etc.* We do not rely on a centralized repository that must be kept up-to-date with information about the state of the world. Not having a single place where state is kept also eliminates bottlenecks and the need to keep data fresh.

Our idea of policy cooperation builds on top of a number of fundamental concepts of distributed computing, *e.g.,* remote procedure call (RPC) [Lyo84a, Lyo84b, BN84], remote method invocation (RMI) [BNOW94], mobile agents, active networks [SN04, TSS+97], *etc.* What we borrow from these techniques is the capability of remote execution [Sta86, SG90a, SG90b] and mobile code.

object-tuple-expression [*condition-expression*]

Figure 4.7: **Policy rules are composed by a object tuple of subject, action, and target, along with an optional conditional expression.**

Policy rules in our system have two parts. The first part expresses the action that a subject is allowed to perform on a target. The second part is an additional list of conditions that must be evaluated before the rule applies (see Figure 4.7). If the conditions evaluate to true, then the rule can take effect, otherwise the action is rejected. The conditions are logical, mathematical, and set expressions on the attributes (such as identity, location, *etc.*) of the controlled objects (such as files, users, *etc.*). Additionally the conditions can include prespecified, node specific, queries. These queries can be evaluated either locally by the node, or remotely by another node specified in the query definition.

Cooperation Procedure

Once a rule is triggered (by a subject performing an action on a target), the policy node starts evaluating the condition expression, processing queries according to their definition. Queries (discussed in more detail in Sections 4.3 and 5.2.5) can take two forms, *(i)* they can have a local definition, that only applies to the node that they have been defined for, or *(ii)* they can be executable statements sent by another node. In either form, queries must specify where they should be evaluated and what their arguments are.

For the query system to work we have assumed that objects can be assigned names that uniquely identify them on every node of the system. However, their attributes can differ from node to node, taking into account the heterogeneity of each application, operating system, *etc.*

Another characteristic of queries is that they themselves can issue subsequent queries, making them recursive in nature. This property can generate chains of queries. When the chain evaluation completes, they return their result to the calling node.[3]

As we mentioned previously, queries can take two forms. The local definition form has the benefit of taking advantage of local knowledge. For example, the network layer has more state information about what constitutes a *"secure connection"* than a user-level application. This eliminates the need for the calling node to be aware of any details of the remote nodes. Yet, there might be times when we might require a more dynamic solution. The second type of queries offers us just that. It enables the policy system to *ship* policy statements to other nodes for evaluation. Of course, to achieve this we must assume that

[3]This property could *open the door* for possible cycles. There are well known algorithms for detecting such cycles, *e.g.*, for deadlock detection and avoidance [CHM83, Elm86, Kna87, MM84, PS85, SG98, Tan87, TvR85], so they will not be discussed furhter in this dissertation.

```
class PolObj
{
        name
}

class User < PolObj
{
        local
}
```

Figure 4.8: **User object representation at the console. It includes the user name, and whether the user is local or not.**

the calling node has knowledge of the object representation of the remote node.

4.3 Example

Permitting users to access data is often dictated by whether or not they connect to the data repository in a secure way (*e.g.,* IPsec, ssh, *etc.*). However, it is not sufficient to only require a secure connection to the data repository, it is important that all the steps leading to that connection are also secure. It is therefore critical to be able to specify security policies that will enforce such constraints (at least all the way to the *borders* of the security domain).[4]

Consider a policy that states *"allow any trusted user to have complete access on any trusted file, provided that they are connected securely"*. This policy can only be enforced with the cooperation of the file system, the console, and the network layer.

Figures 4.8 and 4.9 show the representation of a user object at two different enforcement nodes, namely the console and the network layer. A user that exists in both nodes should

[4]It possible to exercise control only on the nodes that are within the same administrative domain. It is very difficult to provide guarantees about security properties on domains beyond our control.

```
class PolObj
{
        name
}

class User < PolObj
{
        connectedfrom
        protocol
}
```

Figure 4.9: **User object representation at the network layer. It includes the user name, where the user is connecting from, and what protocol is used for the connection.**

have the same identifier, however other user attributes can be different. In this case the user attributes at the console are his name and whether he is local or not, and at the network layer the user attributes are his name, where he is connected from, and his protocol.

To guarantee that a user connection is secure, the network layer is using a recursive query (see Figure 4.10). The query will first check whether this is an edge network node, with a secure connection coming in. If not, it is an internal "hop", and recursively query the next node.

In Figure 4.11, we show the complete policy. Notice that the first part of the condition is a query to the console. In this case we are actually shipping an executable statement to be evaluated by the console. Of course, forming such statement requires knowledge of the object representation at the node where the statements will be evaluated.

```
class Query < PolObj
{
        args
        expr
}

Query sec {
        "secure", "a",
        "a.connectedfrom == \"outside\" and a.protocol == \"ssh\" or
        query(a.connectedfrom, a.name, \"secure\") and
        a.protocol == \"ssh\""}
```

Figure 4.10: **Recursive query, defined at the network layer, that first tests whether the connection is coming from the outside using ssh. If the connection is coming from another inside node, with a secure protocol, it recursively queries it.**

```
allow COMPLETE_ACCESS on TRUSTED_FILES by user = TRUSTED_USER
if query("console", user.name, "user.local == \"local\"") or
   query(user.connectedfrom, user.name, "secure")
```

Figure 4.11: **Cooperative policy that permits securely connected, trusted users, to have complete access to the trusted files.**

4.4 Discussion

In this chapter we introduced the notion of cooperative policy evaluation and described the model of how it is to be carried out. Under our model, a policy statement can be evaluated on multiple system nodes. This enables policy writers to write policies that they were unable to previously express.

We began with the realization that real world systems have a difficulty enforcing complex policies that involve multiple applications. Most previous work has primarily focused on of security policy for distributed systems, such as policy languages, policy abstraction, policy distribution, and policy enforcement.

Our work focuses on policy cooperation among enforcement nodes. In Section 4.2 we explored that other approaches have a rather limited approach to providing policy cooperation. Instead, our solution leverages more general ideas from distributed computing. We combine methods from mobile code and remote procedure calls, to enable any enforcement node to reach its peers, access their state, and request remote computation.

This architecture extends the set of expressible security policies. Our work, much like the initial work on other areas of distributed computing (*e.g.,* active networks, mobile agents, *etc.*), offers a platform for further investigation of the impact of such cooperative policies. This platform can further be used to explore applications of policy cooperation in real systems, like distributed compute farms, financial institutions, and others.

Chapter 5

Applications

In the previous two sections we discussed the notions of consistency in the system wide security policy as well as the need for cooperation between enforcement elements to implement the specified security policy. We have developed two proof-of-concept systems to investigate our ideas.

Our first system, *virtual private services* (VPS) [IBI+03], focuses on how security policies can govern multiple distributed services in a consistent way. Virtual private services address the problem of security policy consistency at the language level by the more traditional approach of a single global policy language with uniform access to all low-level controlled objects. However policies can be evaluated in a coordinated distributed fashion.

Relying on a single language with a uniform and global namespace is not always realistic in decentralized heterogeneous systems. For this reason, our second system, CANON, makes no such assumptions. CANON uses the procedures described in Chapters 3 and 4 to provide policy consistency and cooperative policy evaluation in decentralized heterogeneous systems.

5.1 Virtual Private Services

Large scale distributed applications such as electronic commerce and online marketplaces combine network access with multiple storage and computational elements. The distributed responsibility for resource control creates new security and privacy issues, which are exacerbated by the complexity of the operating environment. To handle policies at multiple locations, the usual tools available (firewalls and compartmented file storage) are used in ways that are clumsy and prone to failure.

Our approach relies on two functional separations. First, we split policy *specification* and policy *enforcement*, providing local autonomy within the constraints of the global security policy. Second, we create virtual security domains, each with its own security policy. Every domain has an associated set of privileges and permissions restricting it to the resources it must must ::use and the services it must perform. Virtual private services ensure security and privacy policies are adhered to through coordinated policy enforcement points.

5.1.1 Separation of Management and Enforcement

The problems we discussed in Sections 4.1 and 4.2 are exhibited existing security architectures, and originate from the independent nature of each service. Applications have different notions of a security policy, perform access control according to that specification, and are oblivious to the security policies of other applications. This often causes configuration problems, which lead to security violations.

Virtual private services are a new approach to these challenges. Global security policies

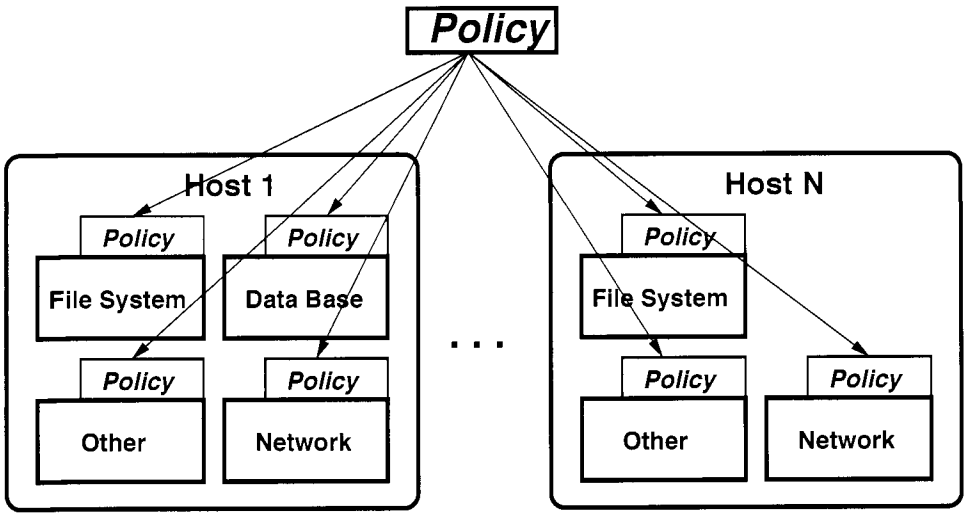


Figure 5.1: **Policy flows from a central specification point to various services. Only the policy rules relevant to a specific service are pulled to that service. No redundant policy state is kept at the access points.**

are specified for services, while enforcement of these policies remains distributed, local to the resource access points. Figure 5.1 shows how policy is managed in this scheme. The policy flows from a central specification point to the various services. Only the policy rules relevant to each specific service are pulled to that service, thus no unneeded policy state is maintained at the various access points. In our architecture, we implement policy *management* with the KeyNote [BFIK99b] trust-management system to express and distribute low-level security policy. Policy *enforcement* is carried out by an augmented host operating system.

Figure 5.2 illustrates virtual private services in the context of a web server application. A CGI script running as part of a web server is restricted to specific subtrees of local and remote file systems, is given a constrained view of a database, and can form network connections only to the machines that host the remote file system and database.

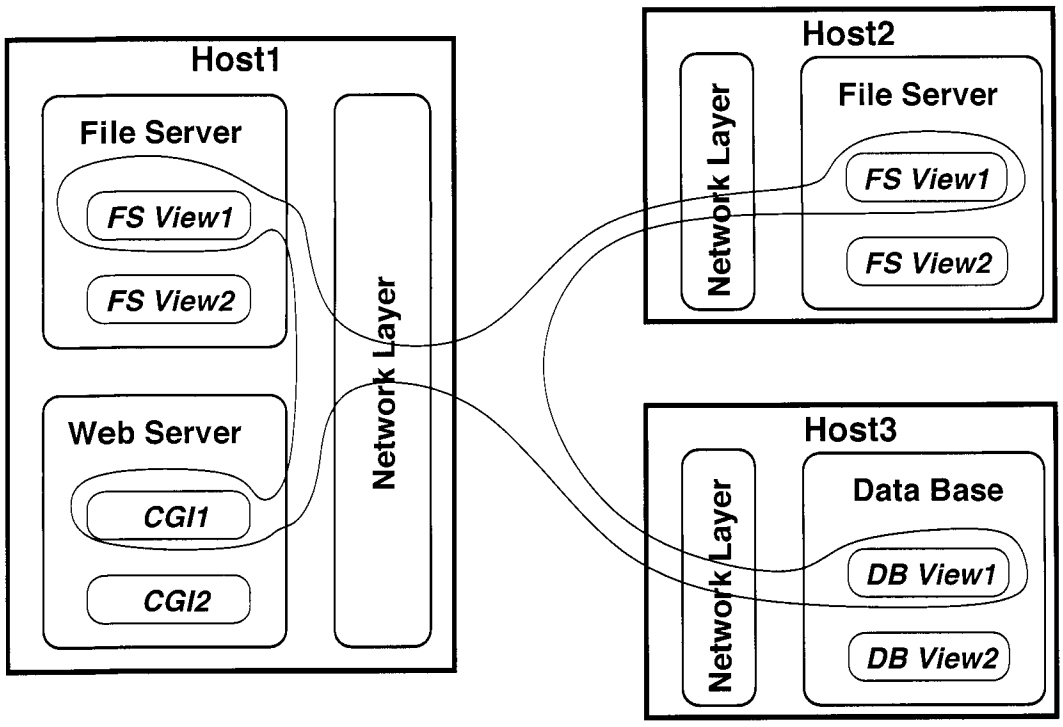


Figure 5.2: **With virtual private services, clients are granted access only to the resources they require to accomplish their task.**

The VPS approach offers multiple benefits. First, it is *scalable* because policy enforcement is done in a distributed fashion by the access points, removing the potential bottleneck of a centralized policy server that must be engaged at policy evaluation/enforcement time.[1] Secondly, it is *flexible,* since maintenance of policies is centralized and coordinated across different applications. Policy modifications automatically propagate to the enforcement points, *simplifying* the task of administration and management of individual devices and applications. Third and finally, the VPS approach allows for policy *consistency*: every service added remains consistent with the central security policy. New services cannot diverge from the existing policies.

[1] A centralized server is required to store and distribute policy. While this could lead to the creation of a bottleneck, policy updates are infrequent.

```
Authorizer: ADMINISTRATOR_KEY
Licensees: USER_A
Conditions: ((app_domain == "db access") &&
  (db_column == "column B") &&
  (permissions == "FULL_ACCESS") &&
  (dst_addr == "Server C") &&
  (src_address == "Host D") &&
  (ipsec_result == "YES")) -> "permit";
```

Figure 5.3: **A simplified representation of the VPS policies for the database example from Section 4.1.**

5.1.2 Policy Translation and Composition

For the architecture to operate across enforcement boundaries and for policy to be globally enforceable, we include "referral" primitives, [KIGS01, IBS01, IBI+03, KIGS03b]; this is simply a reference to a decision made by another enforcement point (typically lower in the protocol stack, but generally at another enforcement point). This primitive allows us to perform policy composition at enforcement time; decisions made by one enforcement mechanism (*e.g.,* IPsec) are made available to higher-level enforcement mechanisms and can be taken into consideration when making an access-control decision.

To accomplish this, all that is necessary is for a channel to propagate this information across enforcement boundaries. In our system, this is done on a case-by-case basis. For example, in our present system IPsec information can be propagated higher in the protocol stack by suitably modifying the Unix `getsockopt(2)` system call; in the case of a web server and SSL, the information is readily available to the web server through the SSL data structures.

```
Authorizer: ADMINISTRATOR_KEY
Licensees: ANY_USER
Conditions: ((app_domain == "net access") &&
    (src_addr == "ALICE") &&
    (dst_addr == "BOB")) -> "permit";
```

Figure 5.4: **Sample policy for allowing network connections between two machines, from Alice to Bob.**

```
Authorizer: ADMINISTRATOR_KEY
Licensees: ANY_USER
Conditions: ((app_domain == "ftp access") &&
   (directory == "/ftpdir/*") &&
   (permissions == "READ") &&
   (dst_addr == "BOB")) -> "permit";
```

Figure 5.5: **Specification for an FTP policy.**

5.1.3 Sample Policies

In Section 4.1 we described an example of a policy for a user accessing a specific column in a database with some additional network constraints. Figure 5.3 shows how such a policy is described. In this example, the administrator authorizes user A to have full access to a database column B, provided they access it on server C coming from host D over IPsec.

In Section 5.1.1 we gave a brief example of a service provided by a CGI script (Figure 5.2). The script requires limited access to the file system (remote and local) and the database, and should not inherit the privileges of the web server. We accomplish this by setting up a distributed policy as seen in Figure 5.8. The first part of the policy guarantees that the script can only connect to either Host2 or Host3 from Host1, the second part will limit file accesses to directories that only contain data for the script, and the last part guarantees that the script will only allow the script to access its own database records.

```
Authorizer: ADMINISTRATOR_KEY
Licensees: WEB_ADM
Conditions: ((app_domain == "fs access") &&
 (directory == "/www*") &&
 (permissions == "FULL_ACCESS")) -> "permit";
```

Figure 5.6: **Policy giving the web administrator full access to the WWW directories.**

```
Authorizer: ADMINISTRATOR_KEY
Licensees: ANY_USER
Conditions: ((app_domain == "web access") &&
   (directory == "/www/webpages/*") &&
   (permissions == "READ") &&
   (dst_addr == "WEB_SERVER") &&
   (dst_port == "80")) -> "permit";
```

Figure 5.7: **Policy allowing any user to access the web server pages.**

The combination of these simple policies assures the properties of the service provided by the CGI script. These sub-policies are independently enforced by the firewall, filesystem, and database server respectively.

Finally, in Figures 5.4, 5.5, 5.6, and 5.7 we give examples of simple policies that define virtual private services for different users and applications. Administrators can customize services in their system by specifying such policies and guaranteeing consistency across all system components.

5.1.4 Evaluation

While the architectural discussion is largely qualitative, some estimates of the system performance are useful. To accomplish this we tested our system with the services for network connection, file system access and web access, defined by the example policies

```
Authorizer: ADMINISTRATOR_KEY
Licensees: CGI1
Conditions: ((app_domain == "net access") &&
 (src_addr == "Host1") &&
 ((dst_addr == "host2") ||
  (dst_addr == "host3"))) -> "permit";

Authorizer: ADMINISTRATOR_KEY
Licensees: CGI1
Conditions: ((app_domain == "fs access") &&
 (directory == "/www/cgi1data/*") &&
 (permissions == "FULL_ACCESS")) -> "permit";

Authorizer: ADMINISTRATOR_KEY
Licensees: CGI1
Conditions: ((app_domain == "db access") &&
 (records == "cgi1records) &&
 (permissions == "FULL_ACCESS")) -> "permit";
```

Figure 5.8: **Set of policies that apply to the CGI script example from Section 5.1.1.**

presented in Section 5.1.3. Even though the sample services are simple, cases operating on a small scale, we believe they provide an adequate picture of the *base* performance of the system.

In our first experiment we wanted to explore the effects that our architecture has on network performance. For this we set up a simple client to consecutively form TCP connections to a server machine. The client and server are interconnected over 100Mbps Ethernet. The average slowdown due to our access control layer was less than 3% (50.4ms vs. 51.8ms). It took 50.4ms to form the connections on a standard OpenBSD system and 51.8ms when we activated the VPS system.

We then simulated a large file transfer over the network by FTP-ing a 100MB file between the server and the client. In this case the our system overhead dropped to less

than 0.5% (11,131ms vs. 11,178ms). This reduction is expected, since the cost imposed by our system (invoked once when the network connection is formed and once when the file is first accessed) is amortized over the entire file transfer.

For our final experiment we used `ab(8)`, the Apache web server benchmarking tool.[2] We ran it for 500 requests with concurrency 1 and 50, the file transferred was 1024 bytes of static HTML. The resulting overhead is about 1%.

5.1.5 A Prototype VPS Implementation

The architecture for virtual private services appears complex. We have added a trust management system, modified host operating systems to control access rights using a policy specification, and added network access control using the same policy specification. We would argue in principle that, since policy dissemination occurs only when needed and the result is cached locally, that the overhead of the trust management system is low. The additional overhead for local enforcement of global policy, however, must be quantified. We used a prototype implementation of virtual private services to perform this evaluation, and were able to limit unwanted unknowns by using a well-understood operating system (4.4 BSD) as a starting point. Section 5.1.5 describes the implementation and Section 5.1.4 the measurements and evaluation.

We used the OpenBSD operating system [ope]. OpenBSD provides well-integrated security features and libraries (an IPsec stack, SSL, KeyNote, *etc.*). Implementations of virtual private services are possible under other operating systems.

[2] `ab(8)` is a tool for benchmarking the performance of the Apache HyperText Transfer Protocol (HTTP) server. It does this by giving an indication of how many requests per second the Apache installation can serve

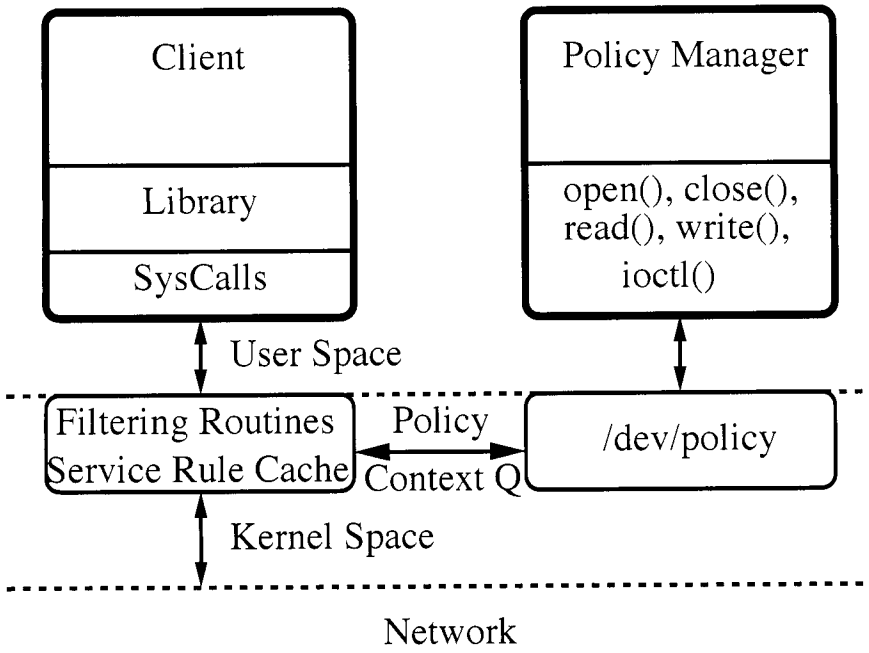


Figure 5.9: **A graphical representation of the system, with all its components. The core of the enforcement mechanism is located in protected kernel space. Each service (*e.g.*, file systems, network layer, *etc.*), has its own filtering routines as well as rule cache for storing policy rules. The policy specification and processing unit lives in user space inside the policy manager process. The two units communicate via a loadable pseudo device driver interface. Messages travel from the system call layer to the user level manager and back using the *policy context queue*.**

Our system has three components:

1. A set of kernel extensions, which implement the enforcement mechanisms at the various access points.

2. A user level daemon process, which implements the centralized policy manager.

3. A pseudo device driver, which is used for two-way communication between the kernel and the policy manager.

Figure 5.9 shows a graphical representation of the system, with all its components. In the following three subsections we describe the various parts of the architecture, their

```
typedef struct  policy_request  policy_request;
struct  policy_request  {
        u_int32_t seq; /*Sequence Number*/
        u_int32_t id;                /*Id*/
        u_int32_t N;  /*Number of Fields*/
        u_int32_t l[N];  /*Field Lengths*/
        char      *field[N];    /*Fields*/
};
```

Figure 5.10: **Policy context data structure**

functionality, and how they interact with each other.

Kernel Extensions

In the UNIX operating system users create outgoing and allow incoming connections, and access the file system using a number of provided system calls. Every system user can use these system calls, it is therefore possible for them to get extensive access to the underlying resources. The reason is that the system call interface provides a coarse grain access control mechanism to the underlying resources [IBS02, Pro03]. This drives the need for some "filtering" mechanism that enables the administrator to limit access. This filtering should be based on a policy that is set by the administrator, and every incoming or outgoing packet as well as file system operation should be subject to it.

Network Access Points To enforce our policy per-packet, we intercept network traffic at the IP layer and pass the packets to our filtering code. We have created two data structures to assist us in this process.

The first data structure, our *rules cache*, contains a set of rules that packets are compared against. If a match is found, the rule is followed to either accept or drop the packet.

The second data structure is the *policy context queue.*

A *policy context* (the C declaration of which is shown in Figure 5.10) is a container for all the information related to a specific packet. We associate a sequence number to each such context and then we start filling it with all the information the *policy manager* will need to make an access control decision.

A request to the policy manager consists of the following fields: a sequence number uniquely identifying the request, the ID of the client the connection request belongs to, the number of information fields that will be included in the request, the lengths of those fields, and finally the fields themselves. This can include source and destination addresses, transport protocol and ports, *etc.* Any credentials acquired through IPsec may also be added to the context at this stage. There is no limit as to the type or amount of information we can associate with a context. We can, for example, include the time of day or the number of other open connections of that user, if we want them to be considered by our decision–making strategy.

As we mentioned already every packet is intercepted at the IP layer and checked against the *rules cache.* If a match is found then the rule is enforced. If no match is found, we enqueue a new request to the *policy context queue.* If we have already enqueued a request for the same class of packets, no further action is necessary. Each entry in the context queue also contains the last packet from that packet flow; if a positive decision is received from the policy manager, the packet is re-queued for processing by the IP stack.

File System Access Points File system access control works in a very similar fashion to network access control. We intercept file system requests and redirect them to our

filtering code.

As in the network case, we have another data structure (with fields relevant for communicating file system information) that holds a set of rules that apply to file accesses. When calls are intercepted we enqueue new requests in the *policy context queue*. The policy manager will receive the request and respond accordingly. Using this technique we can create arbitrary views of the file system, depending on the security policy. This is very much like `chroot(2)` but more like pruning the directory tree of the file system than plainly setting a new root.

Other Access Points There are a number of other resources that can be similarly managed using our architecture, for example memory management and CPU time allocation. For each resource we need to pass the appropriate information between layers using the *policy context queue*. However such controls are beyond the scope of this work and have not been implemented in the current prototype. To enable them, hooks must be added in the memory manager as well as the CPU scheduler, and of course appropriate policies need to be specified in the security manager.

In the next section we discuss how messages are passed between the kernel and the policy manager.

Policy Device

To maximize the flexibility of our system and allow for easy experimentation, we decided to implement the policy manager as a user level process. To support this architecture, we implemented a *pseudo device driver*, `/dev/policy`, that serves as a communication path

```
typedef struct  policy_mbuf     policy_mbuf;
struct  policy_mbuf     {
        policy_mbuf     *next;
        int              length;
        char             data[POLICY_DATA_SIZE];
};

typedef struct  policy_context  policy_context;
struct  policy_context  {
        policy_mbuf     *p_mbuf;
        u_int32_t        sequence;
        char            *reply;
        policy_context  *policy_context_next;
};

policy_context  *policy_create_context(void);
void            policy_destroy_context(policy_context *);
void            policy_commit_context(policy_context *);
void            policy_add_int(policy_context *, char *, int);
void            policy_add_string(policy_context *, char *, char *);
void            policy_add_ipv4addr(policy_context *, char *, in_addr_t *);
```

Figure 5.11: System calls create *contexts* which contain information relevant to that connection. These are appended to a queue from which the policy daemon will receive and process them. The policy daemon will then return to the kernel a decision on whether to accept or deny the connection.

between the user–space policy manager,[3] and the modified system calls in the kernel. Our device driver, implemented as a loadable module, supports the usual operations (`open(2)`, `close(2)`, `read(2)`, `write(2)`, and `ioctl(2)`).

The policy manager reads the device for pending requests in the policy context queue (see Figure 5.11). It then handles the request and returns a new rule to the kernel by writing it to the device, as a result of which the appropriate entry is entered in the rules cache.

The `ioctl(2)` call is used for "house–keeping" tasks. This allows the kernel and the policy manager to re–synchronize in case of any errors in creating or parsing the request messages, and to also flush entries from the rule cache. The approach is similar to [Mog89a, Mog89b].

The Policy Manager

The last component is the policy manager. The policy manager is part of the trusted computing base of our system. It is a user-level process responsible for making decisions, based on policies that are specified by some administrator and credentials retrieved remotely or provided by the kernel, on whether to allow or deny connections.

Policies are initially read in from a file. Addition and removal of policies can be done dynamically. The manager can simply flush one or more entries from the rules cache in the kernel. This way subsequent requests will not match the existing rule set and the policy manager will be queried for the new policy. The manager receives each request from the

[3]The policy manager is part of the trusted computing base and runs as a process in unprotected space outside the kernel.

kernel by reading the `policy` device. The request contains all the information relevant to that connection, as described in Section 5.1.5. Processing of the request is done by the manager using the trust management system, and a decision to accept or deny it is reached. The decision is sent to the kernel, and the manager waits for the next request. While the information received in a particular message is application-dependent, the manager itself has no awareness of the specific application. Thus, it can be used to provide policy resolution services for many different applications, with little or no modification.

Policy Revocation

In an evolving system, security demands and constraints change over time. New users and applications are added, old ones are removed, and services are often moved from one host to another.

As mentioned in the previous section, the system offers policy revocation. Individual entities or sets of entries can be flushed from the rule cache associated with each access point. Subsequent requests will fail to match the cache resident rule set and will be redirected to the policy manager to acquire the new policy. The revocation functionality is provided by the `ioctl(2)` driver calls.

5.2 The CANON Architecture

CANON reduces the "semantic gap" between the natural expression of security policies by users, and the specification of policies in our high-level language (see Figure 5.12). Like other existing security policy specification languages [RZFG01, DDLS01], it abstracts policy beyond the typical low-level ACLs and packet filtering rules.

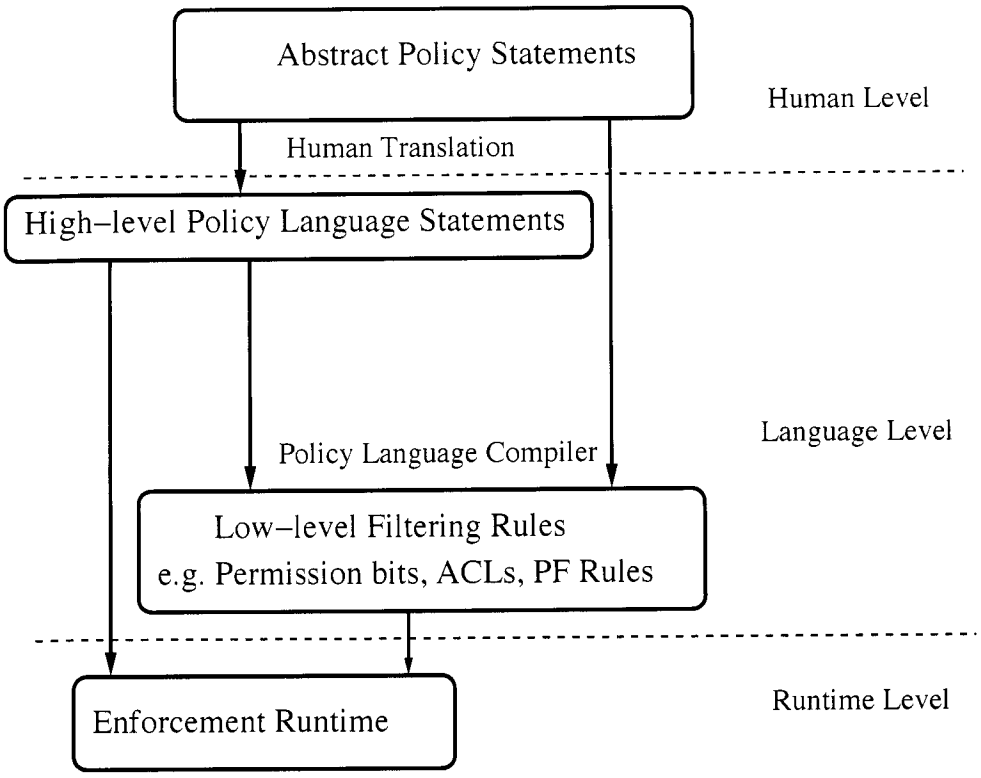


Figure 5.12: **The security policy definition language is designed to express policy similarly to the natural way people do. The high-level specification gets refined to a low-level specification applicable to each type of enforcement element.**

However in contrast to previous work, our system recognizes the heterogeneity of the underlying platforms and creates an interoperability layer to interface with device specific peculiarities. It addresses potential inconsistencies that arise due to the platform specific differences, using the techniques described in Chapter 3. Finally, it enables the definition of cooperative policies described in Chapter 4.

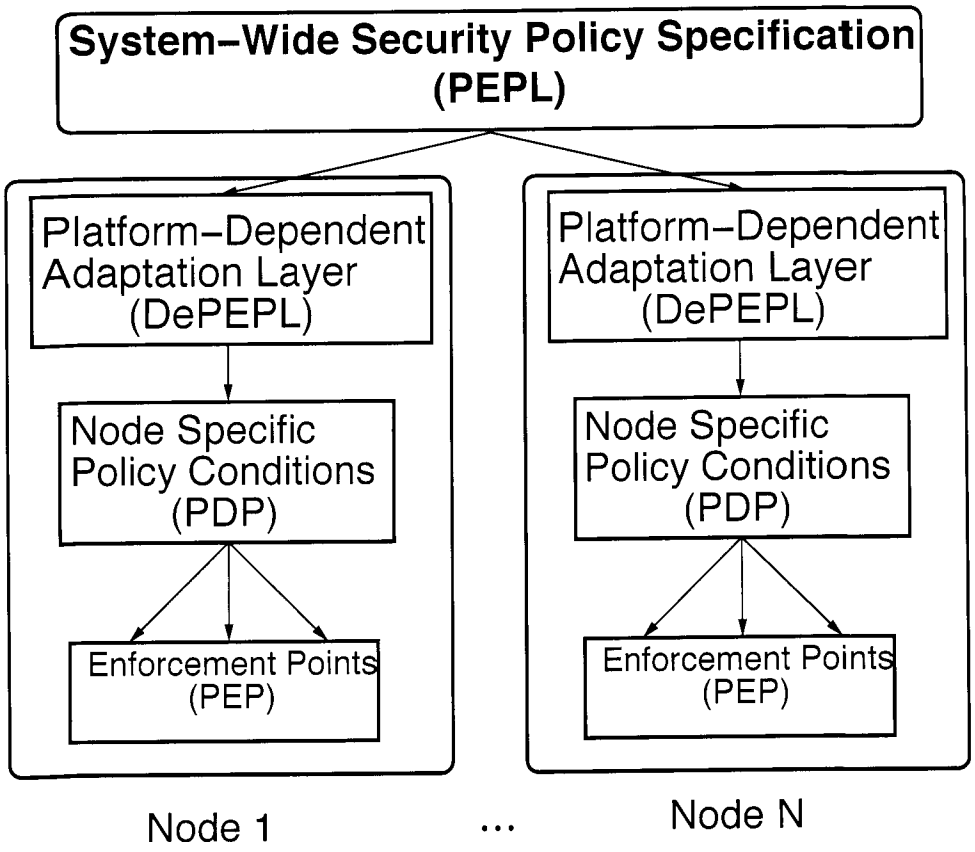


Figure 5.13: **High-level system architecture.**

5.2.1 Structure Overview

There are four main components in the CANON architecture, as shown in Figure 5.13. The term *platform* indicates the type of machine and operating system, while *node* refers to a specific machine. CANON provides:

1. A high-level "policy expression programming language" (PEPL) for administrators to specify security policy.

2. An adaptation layer, or "definition language for PEPL" (DePEPL), that transforms the high-level specification to low-level semantics.

3. Policy decision points (PDPs), execution engines where policy is resolved and decisions are made.

4. Policy enforcement points (PEPs), where access control is exercised.

We will discuss each of them in the following sections.

The standard method of integrating an application into the CANON system is (1) to identify the protected resources controlled and accessed by the application, (2) to logically associate a PEP and a PDP with each instance of those resources,

For example, for the file system implementation, file related operations or system calls, query a PEP that is associated with the file system access control. This request to the PEP includes the uid, and either READ, WRITE, CREATE, or DELETE, and the inode. Similarly, we have built network layer control, along with a PEP, for filtering network accesses. A web server that performs operations with all the privileges (and restrictions!) of the remote user first obtains a token authenticating the remote user, and uses that token to get permission from the PEP before opening each URL.

It is also possible to use CANON to control off-the-shelf, turnkey applications that expose security hooks. In such cases, no explicit PEP exists, and policies delivered to the PDP are simply translated (through DePEPL code executed by the PDP) into calls to the security hooks of the application. Policies that are not supported can signal errors. Of course, there is no way of knowing that the system does not violate security policies internally.

Given a system component that conforms to the CANON model, it remains to describe how policies are specified to the system and which policies apply to objects, and how

policies are enforced.

5.2.2 Policy Specification

(**allow** | **deny**) *action-expr* [**on** *target-expr*] [**by** *subject-expr*] [**if** *condition-expr*]

Figure 5.14: **PEPL statement syntax.**

We designed our policy language system to have a simple grammatical structure, and have a natural human-readable syntax, resembling English used in the policy domain. Figure 5.14 presents the basic syntax of a PEPL statement. Language keywords are denoted by a bold font, choices are enclosed in brackets `()` and separated by |, and optional terms are enclosed in square brackets `[]`. The statement incorporates PEPL expressions, *e.g.*, *action-expr*, which consist of user-defined DePEPL objects, variable bindings, variables, `and`, `or`, and `not`. `and`, `or`, and `not` can be either logical operators and set operators (union, intersection, and complement) depending on context. *condition-expr* has a slightly expanded syntax and includes arithmetic comparison operators. DePEPL objects are treated as primitive tokens by PEPL, and are often constant strings. However, DePEPL hides an arbitrary escape mechanism that allows programmers to expose application- and type-dependent terminology while hiding platform-specific implementation details.

Individual rules expressed in PEPL can either allow or deny any one of a group of agents to perform a set of operations on a set of objects under a set of conditions. It is possible that a particular operation by a given agent on a particular object is subject to more than one rule. These rules may differ in their decision. How do we decide which rule to follow? How do we compose these individual rules into a policy?

PEPL currently gives precedence to rules imposed by higher authority over rules by subordinates. If two rules of equal precedence differ, we give priority to the "narrower" rule.[4] If two rules are of equal precedence, and neither is narrower than the other, we give precedence to "deny." If no rule is found that applies to this operation, we deny the operation.

At first glance this hierarchical model appears restrictive, because if, for example, a CEO says "allow WRITE on FOO by ALL" then no subordinates can deny a WRITE on FOO. If the CEO says "deny WRITE on FOO by ALL" then no subordinates can permit a WRITE on FOO. However, in this system it is still possible for a CEO to cede authority to a subordinate. The CEO imposes no rules relating to the object in question but gives the subordinate authority to add rules (adding rules to, or modifying rules in a policy is an operation that is itself managed by CANON).

It is also possible for a CEO to define the default behavior for a group of subordinates. Suppose the CEO wanted to allow members of the data-center to read log-files. The CEO enters the rule "allow READ on LOG-FILES by DATA-CENTER" under the role of a subordinate (say, the manager of the data-center). The CEO also asserts "allow ADD on FILE-SYSTEM-POLICY by DATA-CENTER-MANAGER" which allows the manager to write rules of equal precedence. Data-center employees will be able to read log-files, but the manager can "deny WRITE on LOG-FILES by BAR" to restrict the ability of just BAR to WRITE on FOO, because "deny" overrides equal precedence rules.

[4]Rule A is narrower than rule B if the agent, operation, resource, and condition specified in A are *each* either equal to, or a subset of, the corresponding term in rule B. The subset relation is by default defined by unpacking the `group` operation in DePEPL, which only works for finite enumerations. It is possible to extend the subset relation by explicit extensions defined in DePEPL – even so, "narrower" is only a partial relation over PEPL rules.

5.2.3 Adaptation Layer

Most real-world distributed systems are heterogeneous. Even the infrastructure of small organizations comprise a multitude of different resources (*e.g.,* printers, firewalls, file servers, *etc.*), that often operate under different operating systems running on diverse hardware. Each of these resource nodes has to be able to exercise access control locally, in order to avoid bottlenecks, but still be bound to the global policy.

Our security architecture supports global security policy for heterogeneous systems. For this, it is necessary to translate the high-level policy specification to the low-level primitives that govern our heterogeneous environment. This task, as we mentioned in Section 5.2.1, is accomplished by our adaptation layer, DePEPL. DePEPL works as an intermediary between the platform and configuration-dependent forms that represent security information, and the types referenced in PEPL, which are similar to object-oriented data-structures. DePEPL defines system concepts as types such that at run-time, information from the system can be interpreted as objects that instantiate those types.

Each type of platform in our system has its own DePEPL specification, and provides a convenient way of hiding device-specific details are of limited interest to users that set up the system policy. We envision DePEPL programmers, experts in specific platforms programming the adaptation layer, and PEPL programmers, administrators setting up the system-wide policy. This is very much like device driver programmers and system programmers. System programmers can write software using the devices, but can ignore device details. When necessary, however, DePEPL exposes details by exporting generic attributes of the underlying devices as parameters to PEPL.

```
class PolObj
{
        name
}

class Host < PolObj
{
}

class User < PolObj
{
        protocol
        enc_alg
}

class Connection < PolObj
{
}

class Query < PolObj
{
        args
        expr
}

Host alice {"Alice"}
User bob {"Bob"}
Connection conn {"Conn"}
Query sec {"secure","a","a.protocol == \"IPsec\" and a.enc_alg == \"AES\""}

TRUSTED-HOST = {alice}
TRUSTED-USER = {bob}
CONNECTION = {conn}
```

Figure 5.15: **Simple example of DePEPL statements. We have defined a number of classes and instantiated objects** `alice`, `bob`, `conn`, **and a query** `sec`. **Then, we defined high-level objects TRUSTED-HOST, TRUSTED-USER, and CONNECTION, and grouped our objects.**

Figure 5.15 shows an example DePEPL specification. We have defined three types of objects, for host, user and protocol, each of which has one attribute, in this case "name".

```
allow CONNECTION on TRUSTED-HOST by u = TRUSTED-USER
if query("network", "secure", u)

((action in CONNECTION) and
 (target in TRUSTED-HOST) and
 (subject in TRUSTED-USER) and
 (s.protocol == "IPsec" and s.enc_alg == "AES")) -> ALLOW
```

Figure 5.16: **Example PEPL policy. Combined with the DePEPL definition from Figure 5.15, it gets translated into a set of conditions, that have meaning on the specific platform and can be resolved by a PDP.**

It is possible to have any number of attributes, for example the user type could also have a user id and a group id, and so on. After we have defined the object types we created instances of these objects, in our example Alice, Bob, and Conn, giving values to their attributes. We then define a *Query* object called *sec*. The first attribute of this Query object is its name "secure", its second attribute is its attributes "a", and its third attribute is the actuall expression that the query will evaluate. Consider now a PEPL policy statement, "allow CONNECTION on TRUSTED-HOST by TRUSTED-USER if u = TRUSTER-USER". Our compiler uses the definitions in Figure 5.15 to transform the policy statement into a set of conditions (Figure 5.16) that have meaning on the target platform. These conditions can now be resolved by a PDP.

5.2.4 Policy Enforcement

Any system contains a large number of potential enforcement points. Typically each resource (*e.g.*, file system, network layer, *etc.*), is protected by a PEP that mediates accesses to protected objects. Each PEP has a small cache of its most recent decisions as a performance optimization. The cache is not needed for the correct operation of the system, and

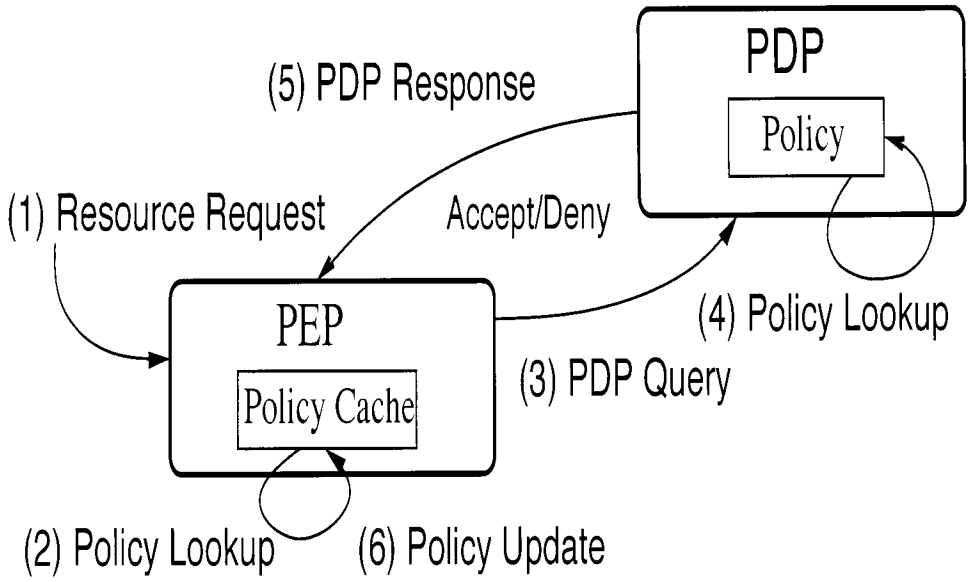


Figure 5.17: **Communication between PDP and PEP.**

the contents can be freely discarded at any time. These caches are flushed when policies are revoked to avoid security violations. Applications (including the OS kernel) request permission from a PEP before performing an operation on a resource (Figure 5.17). The PEP checks its local policy cache to see if the operation is permitted or denied. If there is a cache miss because no cache entry exists, then the PEP queries the corresponding PDP to determine what rules apply in this case. The rules are applied, in order, until a decision is reached. The result, indexed by the triplet <agent/role, operation, object>, is cached in the PEP.

In theory, the PDP has either direct or indirect access to every rule relevant to this object, and must simply match the operation and agent against every rule it knows and determine the most appropriate rule. In practice, a PDP may have an incomplete set of rules locally. In such cases, the PDP searches its local rules. If the local rule-set determines that the correct decision is to deny access, then the PDP returns. If the local

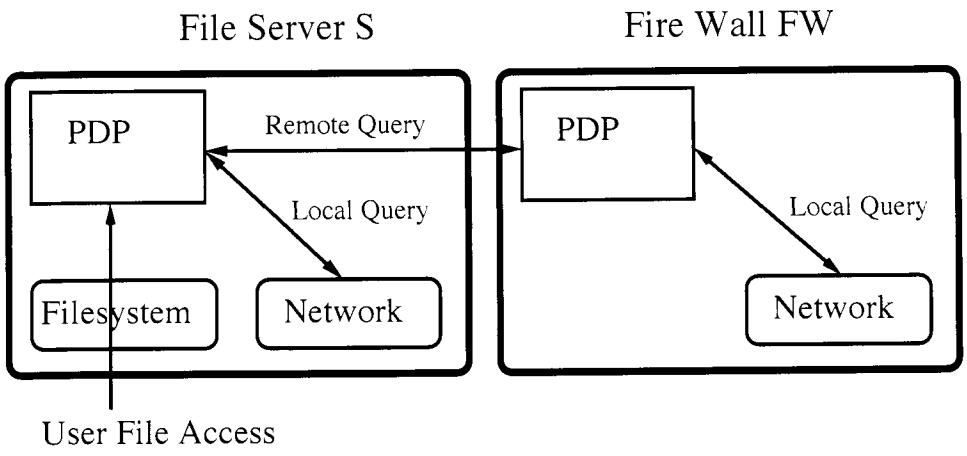


Figure 5.18: **Local and remote queries to gather the information necessary to evaluate a security policy.**

rule-set decides to allow access, then the PDP must search the authorization hierarchy for a deny rule from a higher authority (or, as a performance optimization to cut short the search, for evidence that no higher authority deny rule exists). This algorithm is *conservative*, because it guarantees that the PDP will never incorrectly grant access. The algorithm only incorrectly denies access in the case when an unseen non-local rule overrides the local deny, and should have granted permission. Therefore, because it is conservative and because authenticated rules can be added to PDPs by anyone, we say that the errors it makes are *recoverable*. The errors are recoverable because if the system makes a mistake and denies access, the requesting agent can quickly recover by trying again and passing the authorizing rule to the PDP. Rules are signed and authenticated, so a requesting agent can acquire the rule, pass it to the PDP, and succeed on its next attempt.

5.2.5 Communication between Domains and across Enforcement Layers

Real-world policies often require information from different layers of the system, for example the network and the file system, or even from different nodes, before they can be

evaluated. To decide whether a given operation is allowed, a PDP may need to cross layers of software, physical machines, or even administrative boundaries. Consider a rule that says *"user U may access file F on file server S if U has connected securely over firewall FW"*. For the file server S to enforce this policy it needs to know whether user U has connected securely on firewall FW. This information however is not readily available. More precisely, when the user tries to access the file, the file system has no information about where the user is located. Furthermore it has no way of knowing whether a secure network protocol is being used for transport. Before any decision can be made, the system needs to collect this information.

If the information needed is local, the system uses layer-crossing to extract what it needs. This can be implemented by either modifying, or adding the appropriate system calls. For example, information from the network and the file system layer is obtained by suitably modifying the Unix `ioctl(2)` system call. In the complex case, where information is non-local, our system implements a query mechanism that allows for cross-domain communication between PDPs. When a PDP evaluates a policy statement, for every part of the statement that it cannot decide, it forms a query. This query is issued to another PDP that potentially holds the missing information. This works in a recursive fashion. After the query returns the original statement will be evaluated. The natural question is, how does the original PDP know which other PDP to query? The PDP gets it from the DePEPL definition where this information is explicitly stated. In our example, *secure* will be exactly defined in DePEPL, so the logic inside the PDP will be able to determine who holds the needed information.

In Figure 5.18 we illustrate the process of querying to gather the information necessary

to evaluate the security policy in our example. The PDP will first query the network layer in the file server to determine if the user connection is originating from the firewall. If that is true, it will then issue a remote query to the PDP on the firewall machine asking it whether the user connection to the firewall from the outside is secure. After it has collected this information, the policy can be evaluated, and a decision to accept or deny the user file access can be reached.

5.2.6 Policy Management

Our system supports both positive policy statements, that allow operations and permit access to resources, and negative statements, that restrict permission. Furthermore, multiple users are permitted to create policy. This can lead to potential conflicts when policy is evaluated. There have been a number of proposed solutions for policy resolution in case of conflicts [JSS97, LS97, CC97, LS99]. In our system we have built a straightforward resolution mechanism that gives precedence to policy rules that have been defined by users of higher authority. For example security policy rules specified by the CIO of a company have precedence over rules specified by a system administrator. In the case of rules at the same level of authority we give precedence to negative rules, opting to be conservative.

Positive statements are stored in policy repositories[5] which are local to administration domains and levels of authority, and which can be potentially replicated to resist DoS attacks. It is outside the scope of this dissertation to investigate replication strategies, which have been exhaustively investigated elsewhere [JBH+05].

[5]Policy repositories can be extremely simple directories. Policies are signed and are therefore self-protected. It is possible to provide policy-confidentiality by encrypting each policy statement with the public key of the intended user.

Users can get the policy statements that apply to them and submit them to the relevant PDPs, or the PDPs can look for policy rules on demand, and have no need to know rules that do not apply to them. We call this property "lazy policy instantiation" [KIGS01]. This is intended to allow VPS and CANON to scale well with the number of users, rules, and enforcement points. PDPs may treat policy statements as soft state and thus discard them as soon as storage resources become scarce. Future access will simply trigger the fetching mechanism.

Negative statements are security crucial, and for those we support a *push* model. Under this model, policies issued by administrators are pushed to the relevant PDPs. Furthermore, the already cached decisions that exist in the PEPs controlled by that PDP are flushed. In this manner any subsequent operation will be bound to the updated policy.

5.3 Discussion

In this chapter we presented two systems built as part of the investigation of security policy consistency and cooperation (see Chapters 3 and 4 for the algorithms we developed and implemented) in distributed heterogeneous systems. The first system, VPS, evolved from our work on the distributed firewall [IBS01]. While the distributed firewall focused on efficient distributed enforcement of a centralized security policy, VPS pushes the idea to the more general concept of services. To provide secure services to users, multiple enforcement layers and nodes must work in concert and adhere the global security policy.

CANON in turn, extends VPS with our ideas of policy consistency on heterogeneous systems. CANON utilizes our consistency algorithm, presented in Chapter 3, to detect

inconsistencies in security policy specification that could lead to security failures. Additionally, CANON is capable of sharing and exchanging global state by using remote queries (see Chapter 4), permitting it to cooperatively enforce the global security policy. These two features greatly extend its capabilities over traditional distributed access control systems such as [TJM+99, IBS01, KIGS03b], *etc.*.

Chapter 6

Conclusions

In this dissertation we considered the problem of enforcing security policies consistently across elements of a heterogeneous distributed system. One architectural approach to addressing this problem is to design separable security mechanisms and use multi-level security policy specifications. In such systems, policy rules are written at a level of abstraction intended to correspond closely with the conceptual level at which policies are stated in practice, and are applied through an adaptation layer mapping the rules to each distinct platform. This leaves a semantic gap between the specification of a policy, and the implementation of its enforcement — a gap that is filled by the translation rules.

In the context of such systems, this dissertation identifies a central question: how do we know that the implementation is consistent with the specification? We have broken this down into two distinct issues: preservation of intent, and cross-platform consistency (see Section 3. We have argued that it is impossible to preserve either intention or consistency in the general case, but we have shown several methods for increasing our confidence in the consistency of the system in many cases which we believe will be common.

Since these consistency problems are present in *all* the decentralized heterogeneous systems, and that although still imperfect, our approach is more likely to be consistent than existing systems of this type.

6.1 Results

More specifically, the questions we have answered in this dissertation are:

- *Is it possible to guarantee security policy consistency in decentralized heterogeneous systems?* In its general form this would involve solving the halting problem. Instead we adopted a more pragmatic approach. Security policy is normally defined at a higher level of abstraction on virtual entities and it is the responsibility of the underlying mechanism to enforce it on physical world entities. Having identified this natural layering, we defined a procedure for exhaustively testing whether the high-level policy abstractions and rules are actually maintained by the lower-levels of enforcement.

- *Under what conditions can we guarantee security policy consistency in decentralized heterogeneous systems?* For our solution to be applicable we have a set of requirements. To statically determine consistency we require that all system objects at all layers are defined, as well as knowledge about how objects are mapped between layers. While this might sound like a great burden on the policy writer, most of this information already exists. For example, users and resources have identities, applications and operating systems already use groupings, *etc.* Furthermore, we have

identified cases where consistency cannot be statically defined, *e.g.*, dynamically generated groups. For such cases we have proposed a modification to our consistency procedure, which checks consistency dynamically as the system is running.

- *Is it possible to write and enforce policies that span multiple enforcement nodes and require cooperation between them?* Existing policy systems have taken an isolated approach to security policy, where each system enforces its own part of the global security policy. We showed examples of policies that require cooperation between multiple nodes to be evaluated correctly. Additionally, we demonstrated how we can achieve cooperation using recursive queries between enforcement nodes. Our query system offers the flexibility of letting nodes define their own private definition of a query which takes advantage of the node specific knowledge, but also permits nodes to ship policy statements as dynamic queries to be evaluated remotely.

6.2 Contributions

Our work has a number of additional contributions:

- We introduced the notion of policy consistency for decentralized heterogeneous systems, as defined in Chapter 3.

- A security policy debugger that can assist policy writers to troubleshoot their security policies.

- We have built two systems (see Chapter 5) as *proof-of-concept* of the ideas discussed in Chapters 3 and 4.

6.3 Limitations and Future Work

We made a deliberate decision to focus our study on single-administration hierarchical-domains. These types of environments offer a challenging set of problems, interactions between levels of authority, sharing of state, revocation, *etc.* Solving these problems in single-administration hierarchical-domains is an important step before attempting to tackle issues in multiple, non-hierarchical administrative domains.

Addressing multiple administration domains is a direction to investigate in the future. Such domains pose interesting challenges, for example issues of policy reconciliation [MP02, MP00, WJML04], trust of information on remote state, *etc.* Non-hierarchical domains present their own set of challenges, most notably, which policy takes precedence in case of conflict. Past work has looked into ways of addressing such issues, and we have already discussed this in the dissertation.

Another issue we are not addressing in this dissertation is failure reporting. We decided to be conservative denying access, and assume a higher authority to which we can report errors. There are a number of ways one can address this (see Section 3.3). We could report a policy failure to the agent that performed the access and let them deal with the problem out of band (*e.g.,* talk to the administrator). Another method could be reporting the error to the author of the offending rule, or if multiple rules are involved, to all the authors. There is no shortage of possible "solutions" and we leave investigation of this as future work.

In the context of our consistency algorithm, there are two additional directions we believe are worth exploring to extend our architecture. Firstly, continue this work on ways

to offer some assurances for the preservation of intent of the policy writer. We feel that the methods discussed in Section 3.2 are a good starting point, and can catch most failures in consistency. Secondly, we want to investigate policy failures due to inconsistency in deployed systems. This will allow us to provide a more quantitative evaluation of our consistency algorithm as well as explore the limits of consistency checking.

Finally, the structure of the policy language is an area that is wide open for investigation. We designed our policy language to be grammatically close to English. We believe that we can borrow techniques from natural language processing to push security policy definition even close to the human level.

Bibliography

[aim] AOL Insant Messenger. http://www.aim.com/.

[AMU01] Xuhui Ao, Naftaly Minsky, and Victoria Ungureanu. Flexible Treatment of Certificate Revocation Under Communal Access Control. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2001.

[And01] Ross Anderson. *Security Engineering.* Wiley Computer Publishing. John Wiley & Sons, 2001.

[AR00] Anurag Acharya and Mandar Raje. Mapbox: Using parameterized behavior classes to confine applications. In *Proceedings of the 2000 USENIX Security Symposium*, pages 1–17, Denver, CO, August 2000.

[BBS95] Andrew Berman, Virgil Bourassa, and Erik Selberg. TRON: Process-Specific File Protection for the UNIX Operating System. In *Proceedings of the USENIX 1995 Technical Conference*, New Orleans, Louisiana, January 1995.

[BCD+00] J. Boyle, R. Cohen, D. Durham, S. Herzog, R. Rajan, and A. Sastry. The COPS (Common Open Policy Service) Protocol. Request for comments (proposed standard), Internet Engineering Task Force, January 2000.

[BCG+01] J. Burns, A. Cheng, P. Gurung, S. Rajagopalan, P Rao, D. Rosenbluth, and D. M. Martin Jr. Automatic Management of Network Security Policy. In *DARPA Information Survivability Conference and Exposition (DISCEX I)*, volume 2. IEEE Computer Society Press, June 2001.

[BdVS00] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. A Modular Approach to Composing Access Policies. In *Proceedings of Computer and Communications Security (CCS) 2000*, pages 164–173, November 2000.

[Bel99] S. M. Bellovin. Distributed Firewalls. *;login: magazine, special issue on security*, November 1999.

[BFIK99a] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, pages 185–210. Springer-Verlag Inc., New York, NY, USA, 1999.

[BFIK99b] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote Trust Management System Version 2. RFC 2704, September 1999.

[BFL96] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proc. of the 17th Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, Los Alamitos, 1996.

[BFS98] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance Checking in the PolicyMaker Trust-Management System. In *Proc. of the Financial Cryptography*

'98, Lecture Notes in Computer Science, vol. 1465, pages 254–274. Springer, Berlin, 1998.

[Bib77] K.J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts, April 1977.

[BKRY99] S. Bhatt, A.V. Konstantinou, S.R. Rajagopalan, and Y. Yemini. Managing Security in Dynamic Networks. In *Proceedings of the 13th USENIX Systems Administration Conference (LISA)*, November 1999.

[BL73] D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical Report MTR-2547, Vol. I – III, MITRE Corporation, Bedford, MA, November 1973.

[BLK00] Randeep Bhatia, Jorge Lobo, and Madhur Kohli. Policy evaluation for network management. In *Proceedings of the Nineteenth IEEE Computer and Communication Society INFOCOM Conference*, pages 1107–1116, 2000.

[BMNW99] Yair Bartal, Alain Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. In *Proc. IEEE Computer Society Symposium on Security and Privacy*, 1999.

[BN84] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BNOW94] A.D. Birrell, B.J. Nelson, Susan Owicki, and Edward Wobber. Network Objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 217–230, Asheville, North Carolina, 1994.

[CB94] W. R. Cheswick and S. M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

[CC97] Cholvy and Cuppens. Analyzing consistency of security policies. In *RSP: 18th IEEE Computer Society Symposium on Research in Security and Privacy*, 1997.

[cer] CERT Advisories. http://www.cert.org/advisories/.

[CHM83] K. Mani Chandy, L. M. Haas, and Jayadev Misra. Distributed Deadlock Detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[CL98] M. Carney and B. Loe. A comparison of methods for implementing adaptive security policies. In *Proceedings of the 1998 USENIX Security Symposium*, January 1998.

[CRAG99] P. Calhoun, A. Rubens, H. Akhtar, and E. Guttman. DIAMETER Base Protocol. Internet Draft, Internet Engineering Task Force, December 1999. Work in progress.

[Dam02] Nikodemos Damianou. *A Policy Framework for Management of Distributed Systems*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, February 2002.

[DBSL02] Nicodemos Damianou, Arosha K. Bandara, Morris Sloman, and Emil C. Lupu. A Survey of Policy Specification Approaches, 2002.

[DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder Policy Specification Language. In *Proceedings of Policy Worshop*, pages 18–38, 2001.

[DLSD01] N. Dulay, E. Lupu, M. Sloman, and N. Damianou. A Policy Deployment Model for the Ponder Language. In *Proceedings of the IEEE/IFIP International Symposium on Integrated Network Management*, Seattle, Washington, May 2001.

[Dye88] S. P. Dyer. The Hesiod Name Server. In *Proceedings of the USENIX Winter 1988 Technical Conference*, pages 183–190, Berkeley, CA, 1988. USENIX Association.

[Elm86] A.K. Elmagarmid. A Survey of Distributed Deadlock Detection Algorithms. *ACM SIGMOD Record*, 15(3), September 1986.

[GM03] S. Godik and T. Moses. eXtensible Access Control Markup Language (XACML). www.oasis-open.org/committees/xacml/, 2003.

[Gon99] Li Gong. *Inside Java 2 Platform Security.* Addison-Wesley, 1999.

[GSSC96] M. Greenwald, S.K. Singhal, J.R. Stone, and D.R. Cheriton. Designing an Academic Firewall. Policy, Practice and Experience with SURF. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, pages 79–91, February 1996.

[Gut97] Joshua D. Guttman. Filtering Postures: Local Enforcement for Global Policies. In *IEEE Security and Privacy Conference*, pages 120–129, May 1997.

[GWTB96] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Procedings of the 1996 USENIX Annual Technical Conference*, 1996.

[HBM98] R.J. Hayton, J.M. Bacon, and K. Moody. Access Control in an Open Distributed Environment. In *IEEE Symposium on Security and Privacy*, May 1998.

[HGPS99] J. Hale, P. Galiasso, M. Papa, and S. Shenoi. Security Policy Coordination for Heterogeneous Information Systems. In *Proc. of the 15th Annual Computer Security Applications Conference (ACSAC)*, December 1999.

[Hin99] S. Hinrichs. Policy-Based Management: Bridging the Gap. In *Proc. of the 15th Annual Computer Security Applications Conference (ACSAC)*, December 1999.

[HMM+00] A. Herzberg, Y. Mass, J. Michaeli, D. Naor, and Y. Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *Proceedings of IEEE Symposium on Security and Privacy*, April 2000.

[How97] John D. Howard. *An Analysis Of Security On The Internet 1989 - 1995.* PhD thesis, Carnegie Mellon University, April 1997.

[IBI+03] Sotiris Ioannidis, Steven M. Bellovin, John Ioannidis, Angelos D. Keromytis, and J.M. Smith. Design and implementation of virtual private services. In

Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security, Special Session on Trust Management in Collaborative Global Computing, June 2003.

[IBS01] S. Ioannidis, S. M. Bellovin, and J. M. Smith. Design and implementation of virtual private services. Technical Report MS-CIS-01-13, University of Pennsylvania, 2001.

[IBS02] S. Ioannidis, S. Bellovin, and J. M. Smith. Sub-Operating Systems: A New Approach to Application Security. In *Proc. 10th SIGOPS European Workshop*, pages 108–115, September 2002.

[IKBS00] S. Ioannidis, A.D. Keromytis, S.M. Bellovin, and J.M. Smith. Implementing a Distributed Firewall. In *Proceedings of Computer and Communications Security (CCS) 2000*, pages 190–199, November 2000.

[JBH+05] Flavio Junqueira, Ranjita Bhagwan, Alejandro Hevia, Keith Marzullo, and Geoffrey M. Voelker. Surviving Internet Catastrophes. In *Proceedings of the 2005 USENIX Security Symposium*, pages 45–60, 2005.

[JSS97] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proc. IEEE Computer Society Symposium on Security and Privacy*, pages 31–42, 1997.

[Ker01] A. D. Keromytis. *STRONGMAN: A Scalable Solution to Trust Management in Networks.* PhD thesis, University of Pennsylvania, December 2001.

[KFJ03] L. Kagal, T. Finin, and A. Joshi. A policy language for a pervasive computing environment. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, 2003.

[KIGS01] Angelos D. Keromytis, Sotiris Ioannidis, Michael B. Greenwald, and Jonathan M. Smith. Scalable Security Mechanisms for the Internet. Technical Report MS-CIS-01-05, University of Pennsylvania, April 2001.

[KIGS03a] Angelos D. Keromytis, Sotiris Ioannidis, Michael B. Greenwald, and Jonathan M. Smith. Managing Access Control in Large Scale Heterogeneous Networks. In *Proceedings of the NATO NC3A Symposium on Interoperable Networks for Secure Communications (INSC), The Hague, Netherlands*, November 2003.

[KIGS03b] Angelos D. Keromytis, Sotiris Ioannidis, Michael B. Greenwald, and Jonathan M. Smith. The STRONGMAN Architecture. In *DARPA Information Survivability Conference and Exposition (DISCEX III)*, pages 178–188. IEEE Computer Society Press, April 2003.

[KKK96] Thomas Koch, Christoph Krell, and Bernd Krämer. Policy Definition Language for Automated Management of Distributed Systems. In *Proceedings of the Second International Workshop on Systems Management*, June 1996.

[Kna87] E. Knapp. Deadlock Detection in Distributed Databases. *ACM Computing Surveys*, 19(4), December 1987.

[Koe84] Andrew Koenig. Automatic software distribution. In *USENIX Conference Proceedings*, pages 312–322, Salt Lake City, UT, Summer 1984.

[Lam71] B.W. Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Science and Systems*, pages 437–443, March 1971.

[LBN99] Jorge Lobo, Randeep Bhatia, and Shamim A. Naqvi. A policy description language. In *AAAI/IAAI*, pages 291–298, July 1999.

[LDOW98] Jacob Y. Levy, Laurent Demailly, John K. Ousterhout, and Brent B. Welch. The Safe-Tcl Security Model. In *USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, June 1998.

[LFG99] N. Li, J. Feigenbaum, and B. N. Grosof. A Logic-based Knowledge Representation for Authorization with Delegation. In *PCSFW: Proceedings of The 12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[LPGSF90] M. Larrondo-Petrie, E. Guides, H. Song, and E.B. Fernandez. In D. L. Spooner and Landwehr, editors, *Database Security III*, pages 257–269. Elsevier Science Publishers, Amsterdam, 1990.

[LPI+03] Alexander Levine, Vassilis Prevelakis, John Ioannidis, Sotiris Ioannidis, and Angelos D. Keromytis. Webdava: An administrator-free approach to web file-sharing. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Distributed and Mobile Collaboration*, June 2003.

[LS97] E. C. Lupu and M. S. Sloman. Conflict analysis for management policies. In *Proceedings of the 5th IFIP/IEEE International Symposium on Integrated Network management IM'97, San Diego, CA*, 1997.

[LS99] Emil C. Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852–869, November/December 1999.

[LSDD00] Emil Lupu, Morris Sloman, Naranker Dulay, and Nicodemos Damianou. Ponder: Realizing Enterprise Viewpoint Concepts. In *Proceedings of the 4th International Distributed Object Computing (EDOC2000), Mukahari, Japan*, pages 66–75, September 2000.

[Lyo84a] Bob Lyon. Sun External Data Representation Specification. Technical report, Sun Microsystems, Inc., 1984.

[Lyo84b] Bob Lyon. Sun Remote Procedure Call Specification. Technical report, Sun Microsystems, Inc., 1984.

[Mar97] Damian Marriott. *Policy Service for Distributed Systems.* PhD thesis, Imperial College of Science, Technology and Medicine, University of London, June 1997.

[MBH99] H. Mahon, Y. Bernet, and S. Herzog. Requirements for a policy management system, October 1999. Work in progress.

[MF97] Gary McGraw and Edward W. Felten. *Java Security: hostile applets, holes and antidotes.* Wiley, New York, NY, 1997.

[MK97] David Mazieres and M. Frans Kaashoek. Secure Applications Need Flexible Operating Systems. In *The 6th Workshop on Hot Topics in Operating Systems*, May 1997.

[MM84] D.P. Mitchell and M.J. Merritt. A Distributed Algorithm for Deadlock Detection and Resolution. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 1984.

[MNSS87] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos Authentication and Authorization System. Technical report, MIT, December 1987.

[Mog89a] J. C. Mogul. Simple and flexible datagram access controls for UNIX-based gateways. In *Proceedings of the USENIX Summer 1989 Conference*, pages 203–221, 1989.

[Mog89b] J. C. Mogul. Simple and flexible datagram access controls for UNIX-based gateways. Technical Report 89/4, Digital Equipment Corporation Western Research Laboratory, March 1989.

[Mol95] A. Molitor. An Architecture for Advanced Packet Filtering. In *Proceedings of the 1995 USENIX Security Symposium*, June 1995.

[MP00] P. McDaniel and A. Prakash. Ismene: Provisioning and Policy Reconciliation in Secure Group Communication. Technical Report CSE-TR-438-00, University of Michigan, December 2000.

[MP02] P. McDaniel and A. Prakash. Methods and Limitations of Security Policy Reconciliation. In *2002 IEEE Symposium on Security and Privacy*, pages 73–87. IEEE, May 2002. Oakland, CA.

[MPI+03] Stefan Miltchev, Vassilis Prevelakis, Sotiris Ioannidis, John Ioannidis, Angelos Keromytis, and Jonathan M. Smith. Secure and Flexible Global File Sharing. In *Proceedings of the Annual USENIX Technical Conference, Freenix Track*, June 2003.

[MU98] N. Minsky and V. Ungureanu. Unified Support for Heterogeneous Security Policies in Distributed Systems. In *Proceedings of the 1998 USENIX Security Symposium*, January 1998.

[MWL95] B. McKenney, D. Woycke, and W. Lazear. A Network of Firewalls: An Implementation Example. In *Proceedings of the 11th Anual Computer Security Applications Conference (ACSAC)*, pages 3–13, December 1995.

[NH98] D. Nessett and P. Humenn. The Multilayer Firewall. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, pages 13–27, March 1998.

[NL97] G. C. Necula and P. Lee. Safe, Untrusted Agents using Proof-Carrying Code. In *Lecture Notes in Computer Science, Special Issue on Mobile Agents*, October 1997.

[OGA05] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. MulVAL: A Logic-based Network Security Analyzer. In *Proceedings of the 2005 USENIX Security Symposium*, pages 113–128, August 2005.

[ope] The OpenBSD Operating System. http://www.openbsd.org/.

[Pro03] N. Provos. Improving host security with system call policies. In *Proc. of the 12th Usenix Security Symposium*, Aug 2003.

[PS85] J.L. Peterson and A. Silberschatz. *Operating System Concepts.* Addison-Wesley, Reading, Massachusetts, second edition, 1985.

[PS96] S. Potamianos and M. Stonebraker. The postgres rule system. In J. Widom and S. Ceri, editors, *Active Database Systems - Triggers and Rules for Advanced Database Processing*, pages 44–61. Springer, Berlin, 1996.

[RjL88] M. A. Rosenstein, D. E. Geer jr., and P. J. Levine. The Athena Service Management System. In *Proceedings of the 1988 Winter USENIX Conference*, pages 203–212, Berkeley, CA, 1988. USENIX Association.

[RRSW97] C. Rigney, A. Rubens, W. Simpson, and S. Willens. Remote Authentication Dial In User Service (RADIUS). Request for Comments (Proposed Standard) 2138, Internet Engineering Task Force, April 1997.

[RT74] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[RZFG00] Carlos Ribeiro, Andre Zuquete, Paulo Ferreira, and Paulo Guedes. Security Policy Consistency. Technical Report RT/03/00, INESC, March 2000.

[RZFG01] Carlos Ribeiro, Andre Zuquete, Paulo Ferreira, and Paulo Guedes. SPL: An access control language for security policies with complex constraints. In *Proc.*

of Network and Distributed System Security Symposium (NDSS), February 2001.

[SC98] L.A. Sanchez and M.N. Condell. Security Policy System. Internet draft, work in progress, Internet Engineering Task Force, November 1998.

[SG90a] James W. Stamos and David K. Gifford. Implementing Remote Evaluation. *IEEE Transactions on Software Engineering*, 16(7):710–722, July 1990.

[SG90b] James W. Stamos and David K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[SG98] A. Silberschatz and P.B. Galvin. *Operating System Concepts*. Addison-Wesley, Reading, Massachusetts, fifth edition, 1998.

[SHP88] M. Stonebraker, E.N. Hanson, and S. Potamianos. The POSTGRES Rule Manager. *IEEE Transactions on Software Engineering*, 14(07):897–907, 1988.

[SL93] J. Schönwälder and H. Langendörfer. Administration of large distributed UNIX LANs with BONES. In *Proc. World Conference On Tools and Techniques for System Administration, Networking, and Security Arlington (Virginia)*, April 1993.

[Slo94] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2(4):333–360, December 1994.

[Smi89] Jonathan M. Smith. Practical problems with a cryptographic protection scheme. In Gilles Brassard, editor, *Advances in Cryptology - Crypto '89*, pages

64–73, Berlin, 1989. Springer-Verlag. Lecture Notes in Computer Science Volume 435.

[SN04] Jonathan M. Smith and Scott M. Nettles. Active Networking: One View of the Past, Present and Future. *IEEE Transactions On Systems, Man and Cybernetics, Part C: Applications and Reviews*, 34(1):4–18, February 2004.

[SP98] Ravi S. Sandhu and Joon S. Park. Decentralized user-role assignment for web-based intranets. In *ACM Workshop on Role-Based Access Control*, pages 1–12, 1998.

[SSB+95] D.L. Sherman, D.F. Sterne, L. Badger, S.L. Murphy, K.M. Walker, and S.A. Haghighat. Controlling network communication with domain and type enforce ment. In *Proceedings of the 18th National Information Systems Security Conference*, pages 211–220, October 1995.

[Sta86] James W. Stamos. *Remote Evaluation.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, January 1986. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-354.

[SV00] P. Samarati and S Vimercati. Access control: Policies, models, and mechanisms. *Foundations of Security Analysis and Design*, pages 137–196, 2000.

[Tan87] A.S. Tanenbaum. *Operating systems: design and implementation.* Prentice-Hall, Upper Saddle River, NJ, 1987.

[TEM03] Mary Thompson, Abdelilah Essiari, and Srilekha Mudumbai. Certificate-based Authorization Policy in a PKI Environment. *ACM Transaction on Information and System Security*, 6(4):566–588, November 2003.

[TJM+99] Mary Thompson, William Johnston, Srilekha Mudumbai, Gary Hoo, Keith Jackson, and Abdelilah Essiari. Certificate-based access control for widely distributed resources. In *Proceedings of the USENIX Security Symposium*, pages 215–228, August 1999.

[TMEC02] Mary Thompson, Srilekha Mudumbai, Abdelilah Essiari, and Willie Chin. Authorization Policy in a PKI Environment. In *1st Annual PKI Research Workshop*, 2002.

[TOB98] D. Thomsen, D. O'Brien, and J. Bogle. Role Based Access Control Framework for Network Enterprises. In *Proceedings of the 14th Annual Computer Security Applications Conference*, December 1998.

[TOP99] D. Thomsen, R. O'Brien, and C. Payne. Napoleon Network Application Policy Environment. In *Proceedings of the 4th ACM Workshop on Role-Based Acess Control (RBAC)*, pages 145–152, October 1999.

[TSS+97] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden. A survey of active network research. *IEEE Communications Magazine*, pages 80 – 86, January 1997.

[Tsu92] G. Tsudik. Policy Enforcement in Stub Autonomous Domains. In *Lecture Note in Computer Science 648, ESORICS '92*, pages 229–257. Springer-Verlag, 1992.

[TvR85] A.S. Tanenbaum and R. van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

[VLK00] A. Virmani, J. Lobo, and M. Kohli. Netmon: Network management for the saras softswitch. In *IEEE/IFIP Network Operations and Management Symposium, (NOMS2000)*, pages 803–816, May 2000.

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[WJML04] H.B. Wang, S. Jha, P.D. McDaniel, and M. Livny. Security Policy Reconciliation in Distributed Computing Environments . In *IEEE 5th International Workshop on Policies for Distributed Systems and Networks*, 2004.

[WL93] Thomas Y. C. Woo and Simon S. Lam. Authorizations in Distributed Systems: A New Approach. *Journal of Computer Security*, 2(2-3):107–136, 1993.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software–Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[Woo01] A. Wool. Architecting the Lumeta Firewall Analyzer. In *Proceedings of the 10th USENIX Security Symposium*, pages 85–97, August 2001.

[WSB+96] K. M. Walker, D. F. Stern, L. Badger, K. A. Oosendorp, M. J. Petkac, and D. L. Sherman. Confining root programs with domain and type enforcement. In *Proceedings of the 1996 USENIX Security Symposium*, pages 21–36, July 1996.

[YS96] N. Yialelis and M. Sloman. A Security Framework Supporting Domain Based Access Control in Distributed Systems. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, pages 26–39, 1996.