A New Approach to the Specification of General Computer Security Policies

Ivan Krsul, Eugene Spafford, and Tugkan Tuglular
COAST Techical Report 97-13
COAST Laboratory*
Purdue University
West Lafayette, IN 47907–1398
{krsul,tuglular,spaf}@cs.purdue.edu

Abstract

The notion of Computer Policy *is fundamental to the study of computer security models, the analysis of computer vulnerabilities, the development of intrusion detection tools, and the development of misuse detection tools. Security only makes sense in relation to security policies that specify what is being protected, how it must be protected, who has access to what is being protected, etc. Policies are, however, difficult to write, normally ambiguous, and difficult to understand.*

Existing policy specification models are not suitable for most commercial off the shelf (COTS) systems. The source code for the system may not be available, they operate under constantly changing environments, and the policy requirements may change frequently. Also existing policy models do not capture the temporal characteristics of many real-world computer security policies.

In this paper we present a functional approach to the specification of policies that allows their stepwise refinement, such that at higher levels we deal with abstractions and at lower levels with details. This approach takes advantage of the natural hierarchical organization of computer systems. We show that the this approach can represent the most common policy models and practical real-world policies that are difficult to represent with existing models.

*Portions of this work were supported by sponsors of the COAST Laboratory.

1 Introduction

In [Lev94] Nancy Leveson argues that we have a greater need to develop and validate the underlying principles and criteria for design rather than to develop greater numbers of designs. As we will see in this paper, this notion applies also to the field of computer security policy specification. We need to understand the principles behind the specification of policies, the impact that these may have on the design of operating systems, and the requirements necessary for their enforcement. For example, given an arbitrary policy, what kinds of auditing support do we need for the detection of a violation of that policy? Current approaches such as C2 logging are cumbersome and do not have enough information for the detection of many events (for example, see [CDE^{+}96, Pri98]).

The notion of *Computer Policy* is fundamental to the study of computer security models, the analysis of computer vulnerabilities, the development of intrusion detection tools, and the development of misuse detection tools [Den83, BB96, GS96, KS94]. Security only makes sense in relation to security policies that specify what is being protected, how it must be protected, who has access to what is being protected, etc. Policies are, however, difficult to write, normally ambiguous, and difficult to understand[com96].

Existing policy specification models, however, are either not expressive enough—in the case of models such as MAC, DAC, Lattice—nor simple enough—in the case of formal methods—to be widely used in COTS systems. These systems are usually closed (i.e. the source code is not available for modification), exceedingly complex, or dynamic. Also, existing policy needs do not capture the temporal characteristics of many real-world computer security policies.

Policies cross the boundaries between program specification, implementation, and operation. Existing policy specification models do not cross these boundaries and as such program specification may not take operational constraints into account, implementations may not follow specifications, and operational environments may not conform to these assumed environments at specification and implementation time.

Current operating systems try to enforce policies by policy-specific mechanisms where possible. Some of these mechanisms may fail due to flaws introduced in the design or development, or because of environmental or emergent reasons [AKS96]. In case of failure, the whole system may be affected. Examples of such cases can be found in the computer security literature. [DFW96, MF97, MFS90, MKL+95, Bis95, A+76, LBMC93, BD96, AKS96, Lin75]. Also, policies change faster than mechanisms can.

In this paper we present some solutions to these problems. In particular, we present a functional approach to the specification of policies that allows the stepwise refinement of policies, such that at higher levels we deal with abstractions and at lower levels with details. The model makes the explicit assumption that policies and the value of the system or objects in the system are related.

The proposed model takes advantage of the natural hierarchical organization of computer systems, with systems being composed of objects with attributes. It expresses policies as algorithmic and mathematical expressions that help identify the axiomatic assumptions made in the specification of the policy and that are well suited for the development of program specifications. The model encourages the stepwise refinement of the policy, providing a natural mechanism for mapping upper level policies to detailed specifications. Hence, at higher levels of abstraction we don't need to consider all details and at lower levels of abstraction we only consider the details but not the global picture.

In requiring that the policy specification explicitly list the objects and attributes that are needed to enforce the policy, the model helps to identify the components that are relevant to the policy and hence provides a better understanding of the policy and its impact in the design of the operating system. It also helps identify the information that needs to be logged to detect violations of this policy.

2 Notation

In this section we present the mathematical and algorithmic conventions that will be used throughout the paper. The notation is a simple adaptation of the notations used in [Coh90, Set89].

Function are specified by indicating the parameter types, the return type, and the function body. If the function returns a value, the function name is used as a variable to assign the return value. The format for a function is shown in equation 1.

Conditions and loops are possible and are indicated by the **if** and **do** keywords. The format of conditions and loops is shown in equation 2.

By definition, the body of a loop is not executed if initially the loop condition evaluates to false. One line conditional operations are possible if the operations are followed by a condition. For example, a conditional assignment would have the form:

$$x := S.o \;\; \textbf{if } S.o \text{ is a file;}$$

Statements and conditions are general mathematical expressions where the following special operators are defined:

$\Rightarrow$ ***Some text*** $\Leftarrow$ The text inside the arrows should be considered a comment.

$\wedge$ The short circuit AND logical operator.

$\vee$ The short circuit OR logical operator.

$\forall$ The *for all* operator iterates over all the elements of a set. For example, a loop construct such as "$\forall x \in S$ **do**" would iterate x over all the elements of set S.

$\in$ The *in* operator tests for set membership.

$:=$ Assignment operator.

$::=$ The *definition* operator is used to define functions and terms.

$$
\begin{array}{l}
\textit{Function Name} : \textbf{parameter type} \times \cdots \times \textbf{parameter type} \rightarrow \textbf{return type} \\
\textbf{fun} \quad \textit{Function Name}\,(\textit{parameter name}, \cdots, \textit{parameter name}) ::= \\
\qquad \textit{Function Body Line 1}; \\
\qquad \vdots \\
\qquad \textit{Function Body Line n}; \\
\textbf{nuf}
\end{array}
\tag{1}
$$

$$
\begin{array}{ll}
\begin{array}{l}
\textbf{if} \quad \textit{Condition}\ \textbf{then} \\
\qquad \textit{Condition Body Line 1}; \\
\qquad \vdots \\
\qquad \textit{Condition Body Line n}; \\
\textbf{fi}
\end{array}
&
\begin{array}{l}
\textit{Loop Condition} \quad \textbf{do} \\
\qquad \textit{Loop Body Line 1}; \\
\qquad \vdots \\
\qquad \textit{Loop Body Line n}; \\
\textbf{od}
\end{array}
\end{array}
\tag{2}
$$

$|$ The *such that* operator. It can be used as a qualifier with the $\forall$ operator. For example, the expression "$\forall x \in S \mid x \in E$" would iterate over all the elements in set S that are also in E.

$\triangleright$ A string comparison operator. The expression $x \triangleright y$ would return `true` if the string y begins with string x (x is a substring of y starting with the first character of y).

Other operators, such as $\leq, \neq, =, *, (,), ;$, etc., have their generally accepted meaning.

3 Definitions of Policies

Most definitions of computer security policies have one of the following forms:

1. Policy helps to define what you consider valuable, and specifies what steps should be taken to safeguard those assets [GS96].
2. Policy is defined as the set of laws, rules, practices, norms, and fashions that regulate how an organization manages, protects, and distributes sensitive information, and that regulates how an organization protects system services. [LS90, dod85, SBH+91, com96]
3. Access may be granted only if the appropriate clearances are presented. Policy defines the clearance levels that are needed by system subjects to access objects [dod82, SBH+91].
4. In an access control model, policy specifies the access rules for an access control framework [KC95].

The key concepts in these definitions are value, authorization, access control, protection, and sensitivity of information. Policy is about specifying the level of value acceptable in a system, where value is a subjective and ambiguous measure. Value can be determined by specific factors such as the people that have access (i.e. it is valuable and thus requires top-secret clearance), or abstract and ambiguous factors such as stock market indicators.

A closer look at each of the policy definitions listed above shows that the dominating concept is value. The first definition makes this claim explicitly. The second definition makes the assumption that sensitive information needs to be protected because it has some perceived value. The third and fourth definitions are based on the need for *authorization* and we argue that the development of this set of authorizations is necessary to maintain the perceived value of the system.

Our working definition of policy will be value-base and as follows:

> Policy is defined as a set of rules that define the acceptable *value* of a system, as defined by its owners, as its state changes through time.

As we shall see in later sections, the word *value* inherits all the ambiguities from the previous definitions. However, it also allows us to define a set of scenarios where this ambiguity can be removed almost completely.

4 Proposed Model

In this section we define a model that can be used to define policies as algorithmic and mathematical expressions that, as defined in section 3, can represent policies as a function of value. The model presented allows for the stepwise refinement of policies in a process similar to that of software development in an object oriented framework.

The general idea is to represent policies as a function of the value of the components at a particular level and defining the value of that component as an aggregation of the value of its subcomponents. For example, in object oriented operating systems the value of a system will be the value of the top level object, and the value of that object will be an aggregation of the value of each of its extensions. One of these extensions might be the file systems and the value of the file system will be an aggregation of its individual components or files. The value of a file will be an aggregation of its components or records, etc.

Define a function *Policy* that takes as parameters the state of a system before and after an atomic operation. The function will return a value of `true` or `false` depending on whether the operation has violated policy because the value of the system changed from one state to the next. The *Policy* function then needs a helper function that calculates the value of the system at a particular state. We call this function the *System Value* function and it takes as a parameter the state of a system and calculates the value of that system by aggregating the value of its components. For every component of the system the *System Value* function needs to evaluate an *Object Value* function that will return the value of that object. If the object is composed of sub-objects another evaluation level is needed.

Define a set of objects of interest I to contain all objects that are relevant to our policy. Each set I contains a set of objects $\{o_1, \ldots, o_i, \ldots\}$ where each object can be represented as a n tuple of attributes $\langle a_1, a_2, \ldots, a_n \rangle$ that contain all the information necessary to describe that object completely with respect to the policy.

The execution of a series of instructions, axiomatically considered an atomic instruction with respect to the policy, with a granularity that can be as fine or coarse as necessary, results in a change of the set I as illustrated in Figure 1.

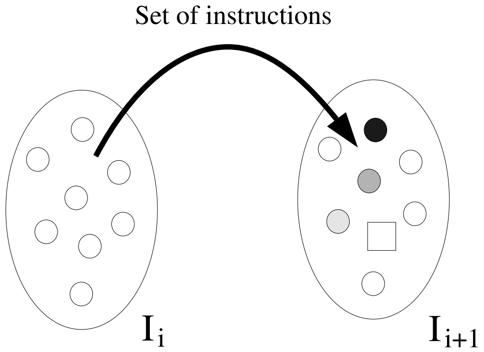


Figure 1. A set of instructions, axiomatically considered an atomic transaction with respect to the policy, causes a change in the set of interest.

As shown in equation 3, a policy can be defined as a function that takes as arguments two numbers, a value function, and two sets of interest (a set of interest before and after the execution of an instruction), and that specifies whether the change in value caused by the instruction is acceptable.

Note that this function permits the definition of growth limits as shown in figure 2. Policies such as *Value* $(I_i) \leq$ *Value* (I_{i+1}) and *Value* $(I_{i+1}) \leq$ *Value* (I_i) are possible by setting a and b to $(1, \infty)$ and $(0, 1)$ respectively[1].

In this paper we limit ourselves to the policy definitions as given by equation 3 and it will be used for all policy functions in this paper unless explicitly stated otherwise. However, other policy definitions are possible and their form will depend on the system and environment. In particular, the policy growth bounds can be made to vary in time as shown by equation 4.

The system value function is an aggregation of the values of the objects in the system. One possible system value function is shown in equation 5.

[1] In practice it is not feasible to include numbers such as ∞ and the policy function must change accordingly.

$$
\begin{array}{l}
\textit{Policy} : \textbf{integer} \times \textbf{integer} \times \textit{Value function} \times \\
\qquad \textbf{set of interest} \times \textbf{set of interest} \rightarrow \textbf{boolean} \\
\textbf{fun} \quad \textit{Policy}\,(a, b, \textit{Value}, I_i, I_{i+1}) ::= \\
\qquad \textbf{if } a * \textit{Value}\,(I_i) \leq \textit{Value}\,(I_{i+1}) \wedge \textit{Value}\,(I_{i+1}) \leq b * \textit{Value}\,(I_i) \quad \textbf{then} \\
\qquad\qquad \textit{Policy} := \text{true}; \\
\qquad \textbf{else} \\
\qquad\qquad \textit{Policy} := \text{false}; \\
\qquad \textbf{fi} \\
\textbf{nuf}
\end{array}
\tag{3}
$$

$$
\begin{array}{l}
\textit{Policy} : \textbf{integer} \times \textbf{integer} \times \textit{Value function} \times \textit{Bounds function} \times \\
\qquad \textbf{set of interest} \times \textbf{set of interest} \rightarrow \textbf{boolean} \\
\textbf{fun} \quad \textit{Policy}\,(a, b, \textit{Value}, \textit{Bounds}, I_i, I_{i+1}) ::= \\
\qquad \textbf{if } \textit{Bounds}\,(a, t) * \textit{Value}\,(I_i) \leq \textit{Value}\,(I_{i+1}) \wedge \\
\qquad\qquad\qquad \textit{Value}\,(I_{i+1}) \leq \textit{Bounds}\,(b, t + \Delta t) * \textit{Value}\,(I_i) \quad \textbf{then} \\
\qquad\qquad \textit{Policy} := \text{true}; \\
\qquad \textbf{else} \\
\qquad\qquad \textit{Policy} := \text{false}; \\
\qquad \textbf{fi} \\
\textbf{nuf}
\end{array}
\tag{4}
$$

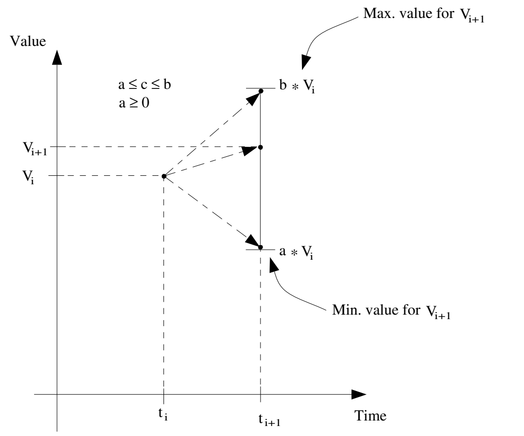


Figure 2. Equation 3 allows the definition of limits for the growth of policy.

$$
\begin{array}{l}
\textit{Value} : \textbf{set of interest} \rightarrow \textbf{integer} \\
\textbf{fun} \quad \textit{Value}\,(S) ::= \\
\qquad \textit{Value} := \sum_{x \in S} v\,(x, S - x) \quad \forall x \in S; \\
\textbf{nuf}
\end{array}
\tag{5}
$$

Note that this function is specific to the policy and other variations may aggregate the values of objects by multiplicative operations, logical operations, algorithmic aggregations, etc. The variation shown in equation 5, however, is general enough for our purposes and will be used for all policy functions in this paper unless explicitly stated otherwise.

Define next the object value function $v\,()$ that takes an object and a set of other related objects as parameters and return a numeric value that represents the value of that object. The *Value()* function that equation 5 refers to is an aggregation of these object value functions. These special object value functions, and their aggregation method, are what allows us to define policies, determine if a policy is ambiguous or unenforceable by a system, and allows us to verify that the protection mechanisms of the system are working appropriately.

Consider the following example: We would like to encode the policy that reads “an email should be deleted only if an identical copy exists. Otherwise it must be saved." Identical emails are those that have the same message-id header. The objects of interest are electronic mails, and the set of interest is the set of all mains in the system.

The object value function v, shown in equation 6, takes as a parameter an electronic mail (the object) and all the other mails in the system (the set of interest) and computes a value that indicates if the system has a duplicate of the mail given.

The parameters for the policy that define the problem of allowing deletions of email only if an identical copy exists are $a = 1$, $b = \infty$, and the value functions defined in equations 5 and 6. For the sets of interest I_i and I_{i+1}, representing the sets of email *before* and *after* any operation that can delete an email, these parameters define the policy $\mathit{Value}\,(I_i) \leq \mathit{Value}\,(I_{i+1})$.

Let the symbol f represent the *Value()* function defined in equation 5. To determine if any operation would result in a violation of policy, an operating system would construct the sets I_i (representing all emails currently held), I_{i+1} (representing all emails left over after the delete operation), and evaluates the function $\mathit{Policy}\,(1, \infty, f, I_i, I_{i+1})$. If this function returns *true* then the operation is allowed by policy and disallowed by policy otherwise.

5 Modeling Existing Policies

Our model must be able to express existing policy models. In this section we show that the most common policies can indeed be represented by our model and that the specification of these models provides insight into models.

5.1 Mandatory Access Control

The Mandatory Access Control (MAC) model requires that subjects have appropriate clearances to access or modify data objects. In this model objects and subjects have associated with them an access level $L \in \{L_1, \ldots, Ln\}$, where there exists a partial ordering $\geq$ on L such that $L_1 \geq L_2 \geq \ \ldots \ \geq L_n$. Each data object and each subject must have an assignment of an access level [KC95].

The policy requires that subjects have an access level that is equal or larger than the level of access of the data object. The set of interest is the set of all subjects and data objects in the system and we require that each subject and data object must have the following attributes defined:

Data Object Attributes:

- ♢ Access level: $d.L$.

Subject Attributes:

- ♢ Access level: $s.L$.
- ♢ An attribute that indicates whether the subject has read a data object: $s.d_i$.

The object value function that can be used to implement the desired policy is shown in equation 7.

The parameters for the policy that specifies this simplified version of the MAC model are $a = 1$, $b = \infty$, and the value functions defined in equations 5 and 7. These parameters define the policy $\mathit{Value}\,(I_i) \leq \mathit{Value}\,(I_{i+1})$.

To see why this works one must start with a system where there are no violations. The system should start in a state where no objects have been accessed and hence the total value for the system, according to equation 5, is zero. As objects are accessed by subjects that have appropriate credentials the total value for the system remains the same. However, if an object is ever accessed by a subject that does not possess the access level required, the value for the system drops and a violation of policy is detected.

There are several important properties that should be noted about this particular model. It first identifies the need to link objects and subjects. It also identifies that the system must enforce the maintenance of the attributes indicated for objects and subjects and, if these are maintained for the system, provides a deterministic mechanism to detect violations of this policy. Note that a practical implementation of this model would only require a list of all objects that have been accessed by the subject, a much weaker condition than keeping ρ attributes, where ρ is the number of objects of the system. This particular policy could also have been specified by storing in the objects a list of subjects that have accessed the object. This information may already be embedded in the audit trail.

If the system can enforce the maintenance of the object and subject attributes then no other access control mechanism other than a process that evaluates the policy given by equation 3 every time a command is

$$
\begin{array}{l}
v : \textbf{object of interest} \times \textbf{set of object of interest} \rightarrow \textbf{integer} \\
\textbf{fun} \quad v\,(o,S) ::= \\
\qquad v := 1; \\
\qquad \forall x \in S \textbf{ do} \\
\qquad\qquad \Rightarrow \textit{Remember that o is not included in S} \Leftarrow \\
\qquad\qquad v := 0 \textbf{ if } x.id = o.id; \\
\qquad \textbf{od} \\
\textbf{nuf}
\end{array}
\tag{6}
$$

$$
\begin{array}{l}
v : \textbf{object of interest} \times \textbf{set of object of interest} \rightarrow \textbf{integer} \\
\textbf{fun} \quad v\,(o,S) ::= \\
\qquad v := 0; \\
\qquad \textbf{if } \ o \text{ is a data object } \textbf{then} \\
\qquad\qquad \forall x \in S \mid x \text{ is a subject } \textbf{do} \\
\qquad\qquad\qquad \Rightarrow \textit{o is an object and, as indicated above, s.o indicates whether s has read o} \Leftarrow \\
\qquad\qquad\qquad v := v - 1 \textbf{ if } \ s.o \neq 0 \ \wedge \ s.L < o.L; \\
\qquad\qquad \textbf{od} \\
\qquad \textbf{fi} \\
\textbf{nuf}
\end{array}
\tag{7}
$$

executed is necessary. If the policy has to be enforced, rather than simply detecting a violation, the system can create a set I_{i+2} that *simulates* the changes to the attributes of the objects in I_i and can evaluate function 3 to determine whether an operation is allowed.

5.2 Discretionary Access Control

Discretionary Access Control (DAC) policies are those where subjects can set the access rights to the objects they own or have permissions to manage.

The set of interest for this policy includes subjects (processes) and data objects (files, sockets, etc.). The attributes that the operating system must maintain for each of the objects in the set of interest are:

Data Object Attributes:

- ◇ For each subject in the system, and for each operation permitted, an attribute indicating whether the subject is allowed to perform the operation on the object: $d.s_i.R$ for reading, $d.s_i.W$ for writing, $d.s_i.C$ for changing attributes, etc.
- ◇ For each subject in the system, and for each operation permitted, an attribute indicating the date the permission bit was set: $d.s_i.R.D$ for reading, $d.s_i.W.D$ for writing, $d.s_i.C.D$ for changing attributes, etc.

Subject Attributes:

- ◇ For each object in the system, and for each operation permitted, an attribute indicating whether the subject has performed the operation on the object: $s.d_i.R$ for reading, $s.d_i.W$ for writing, $s.d_i.C$ for changing attributes, etc.
- ◇ For each object in the system, and for each operation permitted, an attribute indicating the date the subject performed the operation last: $s.d_i.R.D$ for reading, $s.d_i.W.D$ for writing, $s.d_i.C.D$ for changing attributes, etc.

The value function that can be used to implement the desired policy is given by equation 8.

There is a severe limitation to this model, as defined by the policy and value functions in equations 3 and 5, in relation to the delete operation. If a data object is deleted, and assuming that a delete operation was defined, the set of interest *after* the delete operation was completed will not have the data object that was deleted. Hence we cannot compare, as in equation 8, the data object's delete bit for that subject with the subject's delete bit for that object, and we cannot know if the subject had permission to perform the operation. This is a general limitation of this model because the value of the system is defined in terms of the system state at a time and thus information about the past is not available.

v : **object of interest** $\times$ **set of object of interest** $\rightarrow$ **integer**

fun $v(o,S)$ ::=
  $v := 0$;
  if o is a subject **then**
    $\forall x \in S \mid x$ is a data object **do**
      $\Rightarrow$ *There is a violation if the object has been read, the subject does not have permission, and the access took place* after *the policy was set* $\Leftarrow$
      $v := v - 1$ **if** $o.x.R \neq x.o.R \quad \wedge o.x.R.D \leq x.o.R.D$;
      $\Rightarrow$ *There is a violation if the object has been written to, the subject does not have permission, and the access took place* after *the policy was set* $\Leftarrow$ (8)
      $v := v - 1$ **if** $o.x.W \neq x.o.W \quad \wedge o.x.W.D \leq x.o.W.D$;
  $\vdots$
      $\Rightarrow$ *There is a violation if the object has been changed, the subject does not have permission, and the change took place* after *the policy was set* $\Leftarrow$
      $v := v - 1$ **if** $o.x.C \neq x.o.C \quad \wedge o.x.C.D \leq x.o.C.D$;
    od;
  fi
nuf

There are several possible solutions to this problem. The necessary information about the past can be encoded as attributes that the operating system can maintain, a new value function (equation 5) and a new policy function (equation 3) can be defined so that the value of the system can be determined as a function of past system states, or we can redefine deletion so that information about the last known state is kept.

Of all these solutions, the last is appropriate because attribute maintenance is equivalent to logging. Hence, it is always possible to determine the last known attributes of a data object.

5.3 Lattice Structure

A mathematical structure is a lattice if it is a partially ordered set, *S*, and there exist least upper and greatest lower bound operators on *S*. A "poset" implies that the partial ordering relation is reflexive, transitive, and antisymmetric. A least upper bound operator on *S* provides the unique least upper bound of any two elements in *S*. A greatest lower bound operator on *S* provides the unique greatest lower bound of any two elements in *S* [Den83]. This structure is used to define several security policies for computer systems. For instance, the Biba and Bell LaPadula models (see [Amo94, Den83]) use the lattice of security labels. This section will show that our model can detect violation of all policies based on lattice structure.

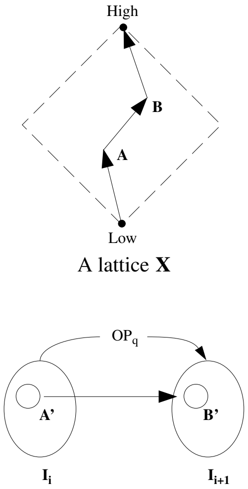
Any lattice has a form similar to that shown in figure 3. In these policies each operation is allowed if the objects occupy levels in the lattice such that there is a partial ordering between them. The direction of the ordering (i.e. whether $A \leq B$ or $B \leq A$) depends on the definition of the operation.

If the system being examined can provide as attributes of objects the operations performed on the objects and the lattice level associated with them, then the system value function can trivially determine if a violation has occurred by performing a simple comparison of the values of objects depending on the operation performed.

5.4 Information Flow

An Information Flow policy is a lattice (*SC*,$\leq$), where *SC* is a finite set of *security classes*, and $\leq$ is a binary relation partially ordering the classes of *SC*. The relation $A \leq B$ means that class A information is lower than or equal to class B information. Information is allowed to flow upwards or within the same class but never downwards [Den83].

Information flows through the application of atomic operations and the granularity of these operations will depend on the policy. Each operation that accesses or writes information will cause information to flow within a class or between classes. In a typical computer system operations are processes (we will refer to



Set of Interest **I**
at time **i** and time **i+1**

Figure 3. Our model can represent lattice based policies by encoding the lattice level as the value of the object.

them as subjects) and information objects are files (we will refer to them as data objects).

We can associate a security class with every subject and data object and for every operation defined in the system we must explicitly state the information flow. For example, the `open()` and `read()` operations causes information to flow from the data object to the subject and hence the subject will inherit the access level of the data object; a `write` or `creat` system call would cause information to flow from the subject to the data object and the latter would inherit the access level of the former.

In the information flow policy model, subjects can access data objects that have the same or lower security class and can write to data objects that have the same or higher security class. Hence, policy violations are 1) when a subject reads a data object that has higher security classification that itself, or 2) when a subject writes to a data object that has lower security classification than itself.

We assume that the operating system can maintain the following attributes for each subject and each data object:

Data Object Attributes:

◇ Current security class: $d.s$.

◇ Incremental change from the last security class: $d.c$.

Subject Attributes:

◇ Current security class: $s.s$.

◇ Incremental change from the last security class: $s.c$.

The *incremental change from last security class* attribute is necessary because the security policy is dependent on both the operation performed (read or write), and the change in security class. From the change we can infer the operation because the subject's security class can only change after a read operation and the object class can only change after a write operation.

The *current security class* attribute needs to be updated by the system according to the following rules:

1. If the operation is a read from a subject (s) to another subject or data object (o), the new access level of s is $s.s := o.s$ **if** $o.s > s.s$.
2. If the operation is a write from a subject (s) to another subject or data object (o), the new access level of o is $o.s := s.s$ **if** $s.s < o.s$.

Hence, a violation of policy occurs if the security class of a subject ever goes up, or the security policy of a data object ever goes down. There is an important exception to this rule. It is concerned with the administrative assignment of a security class to a subject. To change the security class of a user the administrator must lower the security class to zero and then raise it to the new desired level (i.e. delete the user and recreate him).

The value function that can be used to implement the desired policy is shown in equation 9.

Notice that the model presented in this section cannot detect information flow from covert channels. We only consider legitimate channels, which are intended for information flow between subjects, and storage channels, which are data objects shared by subjects [Den83].

Note also that the model will detect multiple violations as the value of the system will decrease every time the policy is violated. A peculiarity of the value

$$
\begin{array}{l}
v : \textbf{object of interest} \times \textbf{set of object of interest} \rightarrow \textbf{integer} \\
\textbf{fun} \quad v\,(o, S) ::= \\
\qquad v := 0; \\
\qquad \textbf{if}\,(o \text{ is a subject} \wedge \\
\qquad\qquad\qquad (o.s = 0 \vee (o.s > 0 \wedge o.c = o.s)))\ \ \textbf{then} \\
\qquad\qquad \Rightarrow \textit{Administrative change of class. We can ignore it.} \Leftarrow \\
\qquad\qquad v := 0; \\
\qquad \textbf{else} \\
\qquad\qquad v := -1\ \textbf{if}\,((o \text{ is a subject} \wedge o.c > 0) \vee (o \text{ is a data object} \wedge o.c < 0))\,; \\
\qquad \textbf{fi}; \\
\textbf{nuf}
\end{array}
\tag{9}
$$

function, as defined here, is that a violation of the policy assumes that the data object or subject involved has been completely compromised. Hence, the granularity of data objects is of critical importance.

For example, if data objects are files in a computer, and a policy violation occurs where a user security class 1 writes to a file with security class 2 (presumably because the access control mechanisms have failed), then the policy violation is noticed and the security class of the file is downgraded to 1 because it is expected that the information in that file now is known by the user of security class 1 and hence it can flow to another user with security class 1.

However, in practice the user may have written only a single byte into the document and the model presented here would downgrade the security class of the rest of the document. This is clearly an undesirable property. The granularity of class labeling should be as fine as necessary. If the user can write a single byte, then files should be labeled a byte at a time. If the user can write in blocks, files should be labeled a block at a time.

5.5 Integrity

Denning writes in [Den83] that:

> "*Information flow models describe the dissemination of information, but not its alteration. Thus, an* Unclassified *process, for example, can write nonsense into a* Top Secret *file without violating the multilevel security policy. Although this problem is remedied by access controls, efforts to develop a multilevel integrity model have been pursued. Biba [Bib77] proposed a model where each object and process is assigned an integrity level, and a process cannot write into an object unless its integrity level is at least that of the object (in contrast, its security level must be no greater than that of the object).*"

Hence, the integrity model can be modeled easily by a simple modification to the information flow model where the directions of the checks are inverted.

5.6 Identification and Authentication

In [Woo94] Charles Cresson Wood mentions the following policies for identification and authentication:

1. All users must be given a unique identity prior to being able to use the company's computer system.
2. All users must have their identity verified prior to being permitted to use the company's computer system.

For the first policy each user must be unique in a computer system (i.e. the list of defined users must not have two similar identities). In this case, the set of interest is composed of identity database entries for all users (i.e. password file). The object of interest is an identity database entry. For every entry in the identity database, the system must maintain an identity string $s.id$.

The value function that can be used to implement this policy is shown in equation 10.

The second policy specified by Wood requires that each subject (process) must have an identity and verification token that is used to determine if the claimed identity is true. Therefore, after logon the process's identity and verification token should match those in

$$
\begin{array}{l}
v : \textbf{object of interest} \times \textbf{set of object of interest} \rightarrow \textbf{integer} \\
\textbf{fun} \quad v(o, S) ::= \\
\qquad v := 0; \\
\qquad \forall x \in S \ \textbf{do} \\
\qquad \qquad v := -1 \ \textbf{if} \ o.id = x.id; \\
\qquad \textbf{od} \\
\textbf{nuf}
\end{array}
\tag{10}
$$

the identity database. The set of interest consists of all subjects (processes) of the system and all identity database entries. For every subject in the system and for every entry of the identity database, the system must maintain two attributes the identity token $s.id$ and verification token $s.ver$.

The value function that can be used to implement the desired policy is shown in equation 11:

The parameters for the policy that specifies this IA model are $a = 1$, $b = \infty$, and the value functions defined in equation 5, and in this section. These parameters define the policy $\mathit{Value}\,(I_i) \leq \mathit{Value}\,(I_{i+1})$. With this modeling of the policies, the value of the system remains the same if the policies are followed.

6 Modeling Policies for COTS Systems

The issue of detecting policy violations is particularly difficult in commercial off the shelf (COTS) operating systems such as UNIX, Windows NT, Mac OS, etc. These operating systems provide a series of protection mechanisms that are inadequate for the enforcement of complex real-world policies and, for the most part, are not customizable when the need to detect policy violations arises.

Recall that in section 1 we mentioned that it may be desirable to define policies whose rigid enforcement may be undesirable. In this section we will give examples of such real-world policies, explain how these policies can be encoded in our model.

The simple formal specification of policies described in this document allows us to explore the creation of computer policies and their impact on other frameworks that depend on these specifications. For example, it may be used to determine if an action taken by a protection mechanism is a violation of policy and may point to areas in these mechanisms that may have potential vulnerabilities.

This model also gives us a better understanding of the impact that ambiguities and inconsistencies in policy specifications have in the modeling of computer security protection frameworks.

6.1 The use of games during business hours

Assume that the CEO of a company reads that it is reported that 100 billion dollars a year in lost productivity is caused by computer games [Gib97]. Area managers are worried that the complete removal of these games may affect morale and hence management decides to institute the policy that the computer games that are installed on local systems can only be used outside business hours.

It may be undesirable to enforce this policy automatically because the notion of business hours may vary among groups in the organization, and there may be circumstances where some employees may be allowed, because of exceptional circumstances, to violate this policy.

The set of interest is all the users in the system. To model this policy using the model described in this document we would require that the operating system maintain for every user (subject) a list of programs executed and the dates and times of execution. Note that the notion of C2 logging satisfies this requirement because the execution of every program is recorded with the time of execution and the name of the user that executed the program.

Subject Attributes:

◇ A set of programs executed: $s.P_i$.

◇ The time and date of every program executed: $s.P_i.T$.

◇ An indication of the type of program executed: $s.P_i.Y$.

The value function that can be used to describe the desired policy is shown in equation 12.

v : **object of interest** $\times$ **set of object of interest** $\rightarrow$ **integer**
fun $\quad v\,(o, S) ::=$
$\quad v := 0;$
$\quad$ **if** o is a process **then**
$\quad\quad v := -1;$
$\quad\quad \forall x \in S \mid x$ is an identity database entry **do** (11)
$\quad\quad\quad v := 0$ **if** $o.id = x.id \ \wedge \ o.ver = x.ver;$
$\quad\quad$ **od**
$\quad$ **fi**
nuf

v : **object of interest** $\times$ **set of object of interest** $\rightarrow$ **integer**
fun $\quad v\,(o, S) ::=$
$\quad v := 0;$
$\quad$ **if** o is a subject **then**
$\quad\quad \forall x \in o.P$ **do**
$\quad\quad\quad v := v - 1$ **if** isBusinessHour $(x.T) \ \wedge$
$\quad\quad\quad\quad x.Y =$ a game;
$\quad\quad$ **od**
$\quad$ **fi**
nuf

(12)

$\quad$ *isBusinessHour* : **time** $\rightarrow$ **boolean**
fun $\quad$ *isBusinessHour* $(t) ::=$
$\quad v := false;$
$\quad \Rightarrow$ *We define business hour to be Monday to Friday from 9 to 4.*
$\quad\quad$ Real *business hours are bound to be more complex.* $\Leftarrow$
$\quad v := true$ **if** $((t.hour > 9 \ \wedge \ t.hour < 4) \ \wedge$
$\quad\quad (t.day \neq$ sunday $\wedge \ t.day \neq$ saturday$))$;
nuf

The parameters that specify this policy are $a = 1$, $b = \infty$, and the value functions defined in equations 5 and 7. These parameters define the policy *Value* $(I_i) \leq$ *Value* (I_{i+1}).

It is impossible for the system to maintain the attribute $s.P_i.Y$ for the general case, as users can install their own programs and disguise them as normal programs. However, it is possible for the system to maintain this attribute for all known games installed in the system.

6.2 Stock Market Operations

So far we concentrated on policies that are concerned with access controls. The model presented in this paper, however, can be used to represent and monitor the adherence to other types of policies so long as there are discrete operations that we must consider atomic, that take a set of interest from one state to another, and that the value of the set of interest can be determined.

Consider, for example, the case of a large financial institution where there exists a policy that specifies that arbitrary financial transactions are allowed in the company so long as all the liquidable assets belonging to the company exceed in value the acquired debts by at least 150%.

Implementation of such a policy in the institution's computer systems can be an expensive proposition, time consuming and error prone. However, if the computer system can maintain the appropriate attributes for the relevant objects—and it is likely that they already do—then it is a simple matter to encode the policy. The set of interest are all the records that describe assets and liabilities (we will call these *financial objects*).

Financial Object Attributes:

- ◇ The type of financial object: $o.t$. Valid values are "asset" and "liability."
- ◇ The value of the financial object: $o.\$$.

The value function that can be used to implement the desired policy is shown in equation 13:

6.3 File access restriction in Java

In the JDK, the settings `acl.read` in `.hotjava/properties` are used to grant limited access to local files. `acl.read` is a path specifying directories that can be read from. This policy can be represented by the model presented in this paper as follows:

An atomic operation will be axiomatically defined as the execution of any method in Java. We assume that the Java interpreter could generate a log of the attributes defined and could generate notifications of method completions.

As shown in equation 14, the policy function takes as arguments a system value function, an object value function, and two sets of interest (before and after the execution of an instruction). The function returns `true` if the policy has not been violated and `false` otherwise.

The policy we specify requires that applications only read or write files that are inside a fixed set of directories specified in an Access Control List (ACL). The set of interest consists of subjects (java applications or applets), the files they read or write, and a set of path specifications that indicate the directories where the applications can read or write:

Element of ACL:

- ◇ Path of Object in ACL: $a.f$.

File Object Attributes:

- ◇ Canonical Path Name: $f.c$.
- ◇ Time of Change of Canonical Path Name (for file object f): $f.t$.

Subject Attributes:

- ◇ File Object Accessed: $s.f$.
- ◇ Time of File Object Access (for subject s): $s.t$.

The value function that can be used to implement the desired policy is shown in equation 15.

7 Modeling Policies that Incorporate the Notion of Time

Many policies incorporate the notion of time as an essential component, or explicitly require that the passage of time be considered. The policy considered in

v : **object of interest** $\times$ **set of object of interest** $\rightarrow$ **integer**

$$
\begin{array}{l}
\textbf{fun} \quad v\,(o, S) ::= \\
\quad v := 0; \\
\quad \textbf{if} \;\; o = o_n \;\textbf{then} \\
\quad\quad \Rightarrow \textit{We only need to perform this step once, hence the test} o = o_n \Leftarrow \\
\quad\quad suml := 0; \\
\quad\quad suma := 0; \\
\quad\quad \forall x \in S \cup \{o\} \;\; \textbf{do} \\
\quad\quad\quad suml := suml + x.\$ \;\; \textbf{if} \;\; x.t = \text{liability}; \\
\quad\quad\quad suma := suma + x.\$ \;\; \textbf{if} \;\; x.t = \text{asset}; \\
\quad\quad \textbf{od} \\
\quad\quad v := -1 \;\; \textbf{if} \;\; suma \leq 2.5 * suml; \\
\quad \textbf{fi} \\
\textbf{nuf}
\end{array}
\tag{13}
$$

$$
\begin{array}{l}
\textit{Policy} : \textit{System Value function} \times \textit{Object Value function} \times \\
\qquad\qquad \textbf{set of interest} \times \textbf{set of interest} \rightarrow \textbf{boolean} \\
\textbf{fun} \quad \textit{Policy}\,(\textit{Value}, v, I_i, I_{i+1}) ::= \\
\quad \textbf{if}\; \textit{Value}\,(I_i, v) \leq \textit{Value}\,(I_{i+1}, v) \quad \textbf{then} \\
\quad\quad \textit{Policy} := \text{true}; \\
\quad \textbf{else} \\
\quad\quad \textit{Policy} := \text{false}; \\
\quad \textbf{fi} \\
\textbf{nuf}
\end{array}
\tag{14}
$$

v : **object of interest** $\times$ **set of object of interest** $\rightarrow$ **integer**

$$
\begin{array}{l}
\textbf{fun} \quad v\,(o, S) ::= \\
\quad v := 0; \\
\quad \textbf{if} \;\; o \text{ is a subject } \textbf{then} \\
\quad\quad \Rightarrow \textit{search ACL to see if some ACL object allows the user's access} \Leftarrow \\
\quad\quad m := 0; \\
\quad\quad \forall x \in S \mid x \text{ is an ACL element } \textbf{do} \\
\quad\quad\quad \Rightarrow \textit{Access is allowed if the file is in the ACL} \\
\quad\quad\quad \textit{and the ACL element was defined before the access} \Leftarrow \\
\quad\quad\quad m := 1 \;\textbf{if}\;\; x \triangleright o.f \wedge o.t > x.t; \\
\quad\quad\quad v := -1 \;\textbf{if}\;\; m = 0; \\
\quad\quad \textbf{od} \\
\quad \textbf{fi} \\
\textbf{nuf}
\end{array}
\tag{15}
$$

section 6.1, for example, requires that the system keep a time-stamp for some attributes.

The model presented in this paper can easily incorporate time by taking it into account at any one of the value functions. We can identify three different kinds of time-dependent policies: those that require a record of the time when an attribute of an object was defined (time-stamps), those that need to be aware of the passage of time (clock-ticks), and those that require knowledge of the time when evaluating the policy functions (evaluation-time).

Time-stamps can be thought of as another object attribute, or the attribute of an attribute. If an object o has an attribute $o.c$, the time at which this attribute was defined can be represented by $o.c.t$ or $o.c\_t$.

Clock-ticks are special because every interval in the clock tick must be considered an atomic operation that must trigger the evaluation of the policy functions. The value of the time can be obtained through a special object in the set of interest that we denote with the symbol τ. This object has special attributes defined that return relevant portions of time. $\tau.h$ returns the hour, $\tau.m$ returns the minutes, $\tau.s$ the seconds, etc.

The following are examples of policies that can be expressed given the assumptions above:

- The use of games during business hours (see section 6.1), as an example of a policy that requires time-stamps.
- Digital time-locks are an example of policies that require clock-ticks in the object value function. The policy states that a particular document cannot be read/written-to/unlocked until a specified time, or until a specified number of clock-ticks have elapsed.
- A policy that reads "The total number of users that can access a resource between noon and midnight should not exceed a threshold," is an example of a policy that would require the use of clock-ticks in the system value function.
- A policy that requires that the value of the system satisfies certain constraints within some time interval (for example during the morning hours) is an example that would require clock-ticks in the policy function.

8 Future Work

In this section we present some of the issues that have not been addressed in this paper but that require more research.

8.1 Development of a Comprehensive Library of Policy Functions

The model presented in this paper is well suited to the development of a comprehensive library of policy functions that can describe a wide variety of common policies for different environments. There are a number of benefits to developing a library of policies, including the public availability of these functions, encouraging the refinement and verification of the functions, etc.

8.2 Limited Policy Violation Prevention

The models and systems presented in this paper can be modified to perform a limited form of policy violation prevention under special circumstances.

Under the assumption that an operation implemented in hardware, firmware, or software is well defined and its implementation conforms to its specification, it is possible to assume that certain conditions will hold after the execution of the operation.

For example, it is possible to assume that a machine instruction will execute and that its execution will have certain well defined effects on the computer system. Under the assumption that the `load` instruction is correctly implemented, we can conclude that the execution of the "`load REG_0 0x00FF`" statement will affect the contents of register `REG_0` and that its value after the execution will be `0x00FF`.

Similarly, we may assume that higher level instructions or system calls, if correctly implemented according to specifications, will have a certain effect on certain objects. For example, under the assumption that the `read` system call in UNIX is correctly implemented, we can assume that the statement that allows user `gollum` to read the file `password` will result in the modification of the `read_by` attribute for the file `password` such that it will contain the value `gollum`.

Hence, it is possible for an operating system to use a mechanism based on our model to theorize the state of the system after an atomic operation, with respect to the policy, and hence evaluate the policy functions for the theorized state and the previous state and determine if a policy violation would occur if the operation is allowed to execute. This constitutes a limited form of policy violation prevention.

A word of caution is appropriate here. In theorizing the result of an operation, the operating system may be making a mistake because the actual implementation of the operation may not adhere to specification. Hence, it is possible for the operating system to deny an operation that would not cause a violation of policy or allow an operation that would violate policy. In both cases, the operation *should not* violate policy but the actual implementation differs from the theoretical result.

In the example given above, the operation `read` on the `password` file may not modify the `read_by` attribute and hence it may not contain the name of the user `gollum`. In this case, the mechanism would not allow the operation to go through because it expects the operation to violate policy even though it does not. Conversely, the `read` operation on the file `/tmp/temp_file` may modify the `read_by` attribute of the `password` file—because of the exploitation of a vulnerability, for example—to contain the user `gollum` even though it should not and the mechanism will allow the operation to continue even though it should not.

Hence, this model could be used to prevent policy violations under the assumption that it is feasible to theorize the result of atomic operations (with respect to policy).

There is a need to provide security mechanisms that do not constitute a single point of failure. Stephanie Forrest et al. argue that "... Many computer security systems are monolithic, in the sense that they define a periphery inside which all activity is trusted. When the basic defense mechanism is violated, there is rarely a backup mechanism to detect the violation." [FHS97].

This model, used as a limited form of policy violation prevention mechanism can be used as an additional layer of protection that prevents the execution of operations that *should not* be executed (regardless of whether their actual execution would violate policy). In this role this model cannot guarantee that operations allowed to execute will not violate policy and the detection of this these events is left to the policy violation detection mechanism.

9 Conclusions

In this paper we have presented a model for expressing policies as algorithmic and formal expressions that can be defined in a process that is akin to the software development cycle, allowing the stepwise refinement of policies. We believe that this model is a significant improvement in the area of policy specification because it starts with the assumption that organizations will use COTS systems and that models for analyzing and specifying policies must take the limitations of these systems into account.

The model is also well suited for the development of a mechanism that can be used for detecting policy violations for policies that can be expressed with our model. We believe that such a system could be implemented readily and could provide real-time detection for most policy violations.

References

[A+76] R.P. Abbott et al. Security Analysis and Enhancements of Computer Operating Systems. Technical Report NBSIR 76-1041, Institute for Computer Science and Technology, National Bureau of Standards, 1976.

[AKS96] Taimur Aslam, Ivan Krsul, and Eugene Spafford. Use of A Taxonomy of Security Faults. In *19th National Information Systems Security Conference Proceedings*, Baltimore, MD, October 1996.

[Amo94] Edward Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall, 1994.

[BB96] Matt Bishop and Dave Bailey. A Critical Analysis of Vulnerability Taxonomies. Technical Report CSE-96-11, Department of Computer Science at the University of California at Davis, September 1996.

[BD96] Matt Bishop and Michael Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, 1996.

[Bib77] Keneth J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESDTR-76-372, The MITRE Corporation, Bedford, MA, April 1977.

[Bis95] Matt Bishop. A Taxonomy of UNIX System and Network Vulnerabilities. Technical Report CSE-95-10, Department of Computer Science at the University of California at Davis, 1995.

[CDE+96] Mark Crosbie, Bryn Dole, Todd Ellis, Ivan Krsul, and Eugene Spafford. IDIOT - Users Guide. Technical Report TR-96-050, Purdue University, September 1996.

[Coh90] Edward Cohen. *Programming in the 1990s*. Springer-Verlag, 1990.

[com96] A Guide to Developing Computing Policy Documents, 1996.

[Den83] Dorothy Denning. *Cryptography and Data Security*. Addison-Wesley Publishing Company, 1983.

[DFW96] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy 1996*, pages 190–200. Princeton University, 1996.

[dod82] DoD 5200.1R, The Department of Defense Information Security Program Regulation, July 1982.

[dod85] DoD 5200.28-STD, Department of Defense Trusted Computer Systems Evaluation Criteria, December 1985.

[FHS97] Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer Immunology. *Communications of the ACM*, 40(10):88–96, October 1997.

[Gib97] W. Wayt Gibbs. Taking Computers to Task. *Scientific American*, pages 82–89, July 1997.

[GS96] Simson Garfinkel and Gene Spafford. *Practical UNIX and Internet Security*. O'Reilly & Associates, Inc., second edition edition, 1996.

[KC95] I-Lung Kao and Randy Chow. Enforcement of Complex Security Policies with BEAC. In *Proceedinngs of the 18th National Information Systems Security Conference*, volume I, pages 1–10. National Institute of Standards and Technology/National Computer Security Center, October 1995.

[KS94] Sandeep Kumar and Eugene Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *17th National Computer Security Conference*, 1994.

[LBMC93] Carl Landwher, Alan Bull, John McDermott, and William Choi. A Taxonomy of Computer Program Security Flaws. Technical Report NRL/FR/5542–93-9591, Naval Research Laboratory, November 1993.

[Lev94] Nancy Leveson. High-pressure Steam Engines and Computer Software. *Computer*, 27(10):65–73, October 1994.

[Lin75] Richard R. Linde. Operating system penetration. In *National Computer Conference*, 1975.

[LS90] Dennis Longley and Michael Shain. The Data and Computer Security Dictionary of Standards, Concepts, and Terms, 1990.

[MF97] Gary McGraw and Edward W. Felten. *Java Security: Hostile Applets, Holes and Antidotes*. John Wiley & Sons, Inc., 1997.

[MFS90] B. Miller, L. Fredrikson, and B. So. An Embirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[MKL+95] Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Akitkumar Natarajan, and Jeff Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical report, Computer Science Department, University of Wisconsin, November 1995.

[Pri98] Katherine Price. Misuse Detection Needs and Auditing System Capabilities (Preliminary Title). Master's thesis, Purdue University, 1998.

[SBH+91] Daniel F. Sterne, Martha A. Branstad, Brian S. Hubbard, Barbara A. Mayer, and Dawn M. Wolcott. An Analysis of Application Specific Security Policies. In *Proceedings of the 14th National Computer Security Conference*, volume I, pages 25–36, October 1991.

[Set89] R. Sethi. *Programming Languages Concepts and Constructs*. Addison–Wesley Publishing Company, 1989.

[Woo94] Charles Cresson Wood. *Information Security Policies Made Easy*. BookMasters, Ohio, 4 edition, 1994.