# A logic of access control

Jason Crampton, George Loizou and Greg O'Shea

*Department of Computer Science, Birkbeck College, University of London, Malet Street, London, WC1E 7HX, England*

e-mail: ccram01@dcs.bbk.ac.uk

October 3, 2000

**Abstract**

The effectiveness of an access control mechanism in implementing a security policy in a centralised operating system is often weakened because of the large number of possible access rights involved, informal specification of security policy and a lack of tools for assisting systems administrators. Herein we present a logical foundation for automated tools that assist in determining which access rights should be granted by reasoning about the effects of an access control mechanism on the computations performed by an operating system. We demonstrate the practicality and utility of our logical approach by showing how it allows us to construct a deductive database capable of answering questions about the security of two real-world operating systems. We illustrate the application of our techniques by presenting the results of an experiment designed to assess how accurately the configuration of an access control mechanism implements a given security policy.

## 1 Introduction

The access control mechanism in a multi-user operating system provides one of the most important security measures employed in commercial environments. However, in practice, access control mechanisms often prove unreliable for enforcing security, because either the requirements are not formally specified or the number of possible access rights is too large for the systems administrator to cope with in the absence of automated tools to determine which access rights should be granted [21]. There are three issues to consider:

- The specification of security, without which we cannot verify the security afforded by an access control mechanism any more than we can talk about the correctness of a program. Specification schemes that pertain specifically to access control have been studied elsewhere, notably in [8] and [24];
- The configuration of the access control mechanism of an operating system, which may involve a large set of objects, such as files and directories, and support many users. Real-world evidence suggests that serious security flaws occur because of errors or omissions made at this level. The primary focus of this paper is to investigate the ability to describe and reason about the implementation of a real-world access control mechanism;
- Whether the state of an access control mechanism meets the requirements of an abstract security policy. In Section 6 we present the results of an experiment designed to assess how accurately a configuration of an access control mechanism implements a given security policy.

The context for our work is multi-user centralised operating systems with discretionary access control mechanisms which support complex applications whose functions and data should only be made available to users in a controlled manner. In contrast, many computers support storage and processing of personal or public data for which access control decisions are straightforward. Such cases are not the primary motivation for our work.

In this paper we describe a new application of modal logic that provides a formal foundation for automated tools that assist in determining which access rights should be granted. Our logic assists in reasoning about the effects of an access control mechanism on the computations performed by an operating system. This makes it possible for the systems administrator of a large computer system, where the number of access rights is overwhelming, to answer some hitherto impractical but important questions, such as:

- can an access right be revoked without reducing the functionality of the system and availability of information to users?
- have all of the access rights needed by a computation been granted, and does granting any such access right have an unforeseen effect that compromises the confidentiality or integrity of the information in the system?

Our logic employs a formal model of an access control matrix, as found for example in the seminal papers of Bell-LaPadula [2], Lampson [11] and Harrison *et al.* [7]. In [2] a model is developed for multilevel secure systems, and an implementation of a security policy that ensures confidentiality of information in such a system is described. In [11] Lampson developed a model of a discretionary access control mechanism based on a protection matrix, and in [7] Harrison *et al.* established complexity results for the safety problem - that is, given a configuration of an arbitrary protection system (modelled as a protection matrix), can a subject acquire a particular access right. While all of these papers inform our work and basic model, our concern is to demonstrate methods that we have developed for reasoning about how primitive access rights and dependencies between them may affect the operation and security of a real-world system.

We regard a *computation* as a finite set of invocations of access rights all of which must be granted by the access control mechanism for the computation to complete successfully. A computation is initiated by a *request* - an event that occurs at the *Trusted Computing Base boundary* [27] - which is characterised by a unique access right.

Some access rights may be invoked in some, but not in all instances of a given request because, for example, of conditional statements in a program. Herein we use the modal logic system **S5**, which deals with possibility and necessity, and which provides a natural way of describing conditional relationships between access rights. Modal logic also provides a convenient foundation for defining abstract objects and types formed out of the primitive objects and types appearing in a deductive database. We envisage that the latter may prove to be important given the large number of facts we must deal with and the potential need to construct a mapping between implementation and abstract security policy. The description of a real-world access control mechanism could involve a large number of facts, and modal logic is relatively simple and efficient to implement by using existing tools.

The remainder of the paper is organised as follows. In Section 2 we present a formal model of an access control mechanism which we will use in providing a semantics to our logic. In Section 3 we describe the syntax, semantics and predicates of our logic. We show that our modal logic can be transformed into first order logic and implemented in Prolog. In Section 4 we define the inference rules of our logic, and introduce a number of queries which we used in experiments against a database of facts and inference rules of our logic. The queries were chosen to provide a suitable means of testing the implementation of our logic, and to illustrate its practical applicability. In Section 5 we describe experiments that involve an implementation of our logic in Prolog and the construction of a deductive database describing two real-world operating systems. The experiments demonstrate that our logic is computationally practical despite the large number of facts needed to describe a real-world operating system. In Section 6 we discuss an experiment where we used a logic program to specify a security policy. We then illustrate how our techniques can be used to compare the configuration of an access control mechanism with such a policy. Finally, in Section 7, we present our conclusions.

Basic theoretic results are deferred to an appendix.

# 2 Preliminaries

We now introduce a formal model of a discretionary access control mechanism (ACM) in a centralised operating system. Our model makes use of the finite state machine formulation of the Bell-LaPadula model [2] which itself incorporates Lampson's protection matrix [11]. We have a finite set, $S$, of active entities, called *subjects*, typically processes or programs in execution; a finite set, $O$, of passive entities, called *objects*, typically files; and a finite set, $R$, of *access rights*. A protection matrix comprises rows representing subjects and columns representing objects where an entry in the protection matrix, $[s, o] \subseteq R$ see [11], indicates the set of access rights which subject $s$ has to object $o$.

Rather than use a protection matrix in our model, we employ a set of triples, $M \subseteq O \times S \times R$, to model an ACM where $(o, s, r) \in M$ if, and only if, (iff) $r \in [s, o]$. The ACM permits a given operation $(o, s, r)$ to succeed if, and only if, $(o, s, r) \in M$. We refer to $M$ as the *state* of the ACM.

The set $F = O \times S \times R$ models all potential access requests to an ACM. At any point in time, a finite number of access rights will be invoked, and the set $A \subseteq M$ models this *active* or *actual* set of functionality.

We now introduce a definition of a relation that will be used to model dependencies between triples in $F$. In Section 3 we introduce the Prolog predicate `needs` to represent members of this relation, and which will form the basis for inferring dependencies between access rights.

**Definition 2.1** *The reflexive relation $N_R \subseteq F \times F$ describes the dependencies between triples in $F$. For a triple, $f$, we define $c_f = \{g : (f, g) \in N_R\}$.*

Note that for all $f$, $f \in c_f$ by the reflexivity of $N_R$.

**Definition 2.2** *Let the set $Q \subseteq F$ be the set of* requests *performed by an operating system. Each request, $q \in Q$, is associated with a* computation, *$c_q \subseteq F$. The request $q$* completes successfully *if $c_q \in M$.*

An example of a request is a command entered by a user at a terminal, to execute a specific program. For example, the command `who` entered at a UNIX terminal would be modelled as a request by the user process to execute the `/bin/who` file. The command `ls /homes/mydir` would be modelled as two requests, the first being a request by the user process to execute `ls`. Assuming that request completed successfully, the second is a request by the `ls` process to read the `/homes/mydir` directory.

Since $Q \subseteq F$ we can use the deductive machinery of our logic consistently, whether we are reasoning about the relationship between a request and a set of triples, or the relationship between a triple and a set of triples. If there is no suitable triple, $f \in F$, that is convenient for representing a given request, then we can easily introduce a nonce triple into $F$ to represent the request; this may involve the introduction of a new element into either the set $O$ or the set $R$.

**Definition 2.3** *For a given set $F = O \times S \times R$, the* state *of an operating system is given by the ordered triple $k = (F, A, M)$, where $A, M \subseteq F$, and $k \in K$, the set of states of the operating system.*

**The operating system model** Following a similar approach to the seminal papers in this field [2, 7, 11, 15, 16], we model an operating system as a deterministic finite state machine $DM(F, C, FS, W, k_0)$, where

- $F = O \times S \times R$ is fixed,

- $C = \{\texttt{invoke}, \texttt{desist}, \texttt{grant}, \texttt{revoke}\}$,
- $FS = \{\texttt{accept}, \texttt{reject}\}$,
- $W \subseteq C \times F \times FS \times K \times K$ is the set of state transitions, and
- $k_0 = (F, A_0, M_0)$, with $A_0 \subseteq M_0$, is the initial state of the finite state machine. (Typically $A_0 = \emptyset$.)

Table 1 shows the state transitions for an element $(c, f, fs, k, k') \in W$, where $k = (F, A, M)$ and $k' = (F, A', M')$.

| Command | $f \in M$ | $f \in A$ | Output | $k' = (F, A', M')$ |
|---|---|---|---|---|
| invoke($f$) | T | T | accept | $k' = k$ |
| invoke($f$) | T | F | accept | $M' = M,\ A' = A \cup \{f\}$ |
| invoke($f$) | F | F | reject | $k' = k$ |
| desist($f$) | T | T | accept | $M' = M,\ A' = A \setminus \{f\}$ |
| desist($f$) | T | F | accept | $k' = k$ |
| desist($f$) | F | F | accept | $k' = k$ |
| grant($f$) | T | T | accept | $k' = k$ |
| grant($f$) | T | F | accept | $k' = k$ |
| grant($f$) | F | F | accept | $M' = M \cup \{f\}$, $A' = A$ |
| revoke($f$) | T | T | accept | $M' = M \setminus \{f\}$, $A' = A \setminus \{f\}$ |
| revoke($f$) | T | F | accept | $M' = M \setminus \{f\}$, $A' = A$ |
| revoke($f$) | F | F | accept | $k' = k$ |

Table 1: State transitions of $DM(F, C, FS, W, k_0)$

A subject issues an `invoke` command for an access right when it attempts to access an object, and eventually, once it has finished with the object, it issues the `desist` command. In both events, the state, $k$, of the operating system is changed by the introduction or removal of a triple in the set $A$. The `invoke` command, when applied to a triple $f \in F$, introduces $f$ into $A$ if, and only if, $f \in M$.

State transitions that involve the introduction or removal of a given access right from $M$ are modelled by the `grant` and `revoke` commands, respectively. Removing a triple $f \in M$ simultaneously removes that same triple from $A$ if $f \in A$. Thus $f \in A$ implies $f \in M$. This is in contrast to those real-world operating systems that check access rights only at the time when a file is opened. We observe that, by construction, $A \subseteq M$ implies $A' \subseteq M'$. Hence if $A_0 \subseteq M_0$, $A \subseteq M$ for all states $k = (F, A, M)$ of $DM$.

The set $FS = \{\texttt{accept}, \texttt{reject}\}$ is the set of outputs. A sequence of elements $(c, f) \in C \times F$, represents a computation of the operating system and forms the input string. A sequence of elements $fs \in FS$ forms the output string. The computation completes successfully if, and only if, the output string is a sequence of "`accept`"s.

The Multics kernel project, which sought to improve the security of the Multics operating system, identified a number of important dependencies between objects in an operating system [23]. Of these dependencies, naming and object typing are relevant to access control, but in this paper we deal only with naming. We have developed a more elaborate form of our logic, involving various aspects of object-orientation, in which object typing is a central theme. We will present this work in a future publication.

We do not want to clutter our logic with unnecessary devices for operating upon complex names, nor do we wish to embed the syntax of any particular naming system in our logic. Therefore, we introduce only a simple naming scheme that can serve as the basis for modelling more sophisticated naming schemes. In the following definition we make use of the Kleene star notation (*) indicating the concatenation of 0 or more strings from a language [13].

**Definition 2.4** *Let* $\Sigma_0 = \{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9\}$ *and* $\Sigma_1$ *be the set of separator symbols. We define the set of names to be* $\Sigma^*$*, where* $\Sigma = \Sigma_0 \cup \Sigma_1$.

Every member of $O$, the set of objects, is uniquely identified with a name. The following definition introduces the concept of a binary relation over names, which we will use to represent a hierarchy of objects in $O$.

**Definition 2.5** *We define a digraph* $G = (O, E)$*, whose nodes are the members of* $O$ *and whose directed edges (arcs) belong to the set* $E \subseteq O \times O$*. It is assumed hereafter that every node in* $G$ *(except the root) has exactly one direct parent. Hence* $(x, y) \in E$ *if and only if* $x \in O$, $y \in O$, $x \neq y$*, and* $x$ *is the direct parent of* $y$ *in the name space of a real-world operating system.*

**Example 2.1** *Let*

$$O = \{\text{"/"}, \text{"/a"}, \text{"/aa"}, \text{"/aa/b"}\} \text{ and } E = \{(\text{"/"}, \text{"/a"}), (\text{"/"}, \text{"/aa"}), (\text{"/aa"}, \text{"/aa/b"})\}.$$

*We observe that the interpretation of "/aa" relative to "/a" is unambiguous because* $(\text{"/a"}, \text{"/aa"}) \in E$.

**Definition 2.6** *Let* $D = \{F, C, M, FS, Q, N_R, G\}$*. Then* $D$ *is the set of sets and relations constituting our formal model.*

It is easy to see that $D$ is finite and can be described by a string of finite length, because $D$ comprises finite sets and (binary) relations over those sets.

# 3 The Logic Framework

We use the modal logic system **S5** as a basis for our logic of access control and demonstrate that it can be transformed into a proper subset of first-order logic. We employ certain restrictions to our logic which ensure that its transformation (into first order logic) is Turing computable and that efficient implementation through logic programming tools is feasible [26]. Specifically

- we avoid the use of function variables;
- we have a finite domain of discourse and a finite number of predicates;
- we adopt the *Closed World Assumption* (CWA), whereby any facts not explicitly stated as true are assumed to be false. In particular, any triples not explicitly granted are assumed to be forbidden;
- we avoid the use of disjunctive assertions and rules and restrict ourselves to Horn clauses and positive ground literals (facts);

The elements of our logic are given below.

- *Connectives* - negation (), conjunction ($\wedge$) and implication ($\leftarrow$) of classical propositional calculus
- *Constants* - members of the sets $O$, $S$ and $R$
- *Variables* - range over the sets in the domain of discourse. The variables $o$, $s$, $r$, $q$, subscripted or otherwise, range over the sets $O$, $S$, $R$ and $Q$, respectively, and the variables $x$, $y$ and $z$ range over the set $F$.

- A *term* is either a variable or a constant. A term that contains no variables is a *ground* term. Where a term stands for a variable, the variable is assumed to be universally quantified over the entire formula.
- A *predicate form* is the application of the appropriate number of terms to a *predicate constant* of arity, say $k$, denoted by $P(t_1, \ldots, t_k)$.
- An *atom* is a predicate form. An atom containing no variables is a ground atom or a *fact*.
- A *literal* is an atom or the negation thereof. A literal that contains the connective $\neg$ is a *negative literal*, otherwise it is a *positive literal*.

We assume the rule of *modus ponens*. Namely, if $L$ and $K \leftarrow L$ are theorems, then so is $K$.

### 3.1 The modal logic system S5

We graft modal elements onto our first order logic to form the system **S5**.

- The characteristic axiom $L \leftarrow \Box L$ of modal logic, where the monadic operator $\Box$ is read "necessarily". If $L$ is an atom, then so is $\Box L$, and if $L$ is a literal, then so is $\Box L$.
- The axiom $\Box\Diamond L \leftarrow \Diamond L$ which characterises **S5**, where the monadic operator $\Diamond$ is read as "possibly". It is defined in terms of $\Box$ through the equivalence $\Diamond L \equiv \neg\Box\neg L$. Informally this can be interpreted as,"something is possibly true iff it is not necessarily false". We do not make explicit use of the operator $\Diamond$ in the remainder of this paper.
- A *formula* is recursively defined as follows:
  - all literals are formulae;
  - if $L$ and $K$ are formulae, then so are $L \wedge K$ and $L \leftarrow K$;
  - if $L$ is a formula, then so is $\Box L$.
- We make the following two observations.
  - **S5** has a law of reduction for repeated instances of the modal operators and hence, given a positive literal, $L$, there are only six distinct modalities in **S5**, namely

    $$L,\ \Diamond L,\ \Box L,\ \neg L,\ \neg\Diamond L,\ \text{and}\ \neg\Box L,$$

    In particular $\Box L \equiv \Box\Box L$.
  - **S5** has the following theorems [9].

    $$\Box(L \wedge K) \equiv \Box L \wedge \Box K \quad \text{and} \quad \Box(K \leftarrow L) \equiv \Box K \leftarrow \Box L$$

- A *rule* is a formula of the form

  $$L \leftarrow L_1 \wedge L_2 \wedge \ldots \wedge L_n,$$

  where $L$ and $L_i$, $1 \leqslant i \leqslant n$, are positive literals. The literal $L$ is the *head* of the rule, and the remaining literals form the *body* of the rule. The rules have no negative literals in their bodies, and are therefore *Horn clauses*. The second of the observations in the preceding paragraph enables us to rewrite a formula of the form $\Box(L \leftarrow L_1 \wedge L2 \wedge \ldots \wedge L_n)$ as $\Box L \leftarrow \Box L_1 \wedge \Box L_2 \wedge \ldots \wedge \Box L_n$.

## 3.2 Semantics of modal logic

The semantics of modal logics do not permit a simple mapping onto the domain $\{\mathtt{T}, \mathtt{F}\}$. In particular, the interpretation of the formula, $\Box L$, is that $L$ is *necessarily* true.

An effective basis for giving an interpretation to modal formulae is with reference to Kripke's notion of *possible worlds* [26]. Intuitively, the notion of possible worlds is that there can be several possible interpretations of a formula, each of which is said to occur in a particular *world* (or *point*). The set $W$ of possible worlds is called the *universe*, and an accessibility relation $AR \subseteq W \times W$ is defined over the universe.

The semantics of **S5** are given with reference to an **S5**-model $\mathcal{M}(W, AR, D, V)$, where

- W is a non-empty universe of possible worlds;
- $AR$ is an equivalence relation over $W$ (thus ensuring the validity of the characteristic axioms of **S5** [9]);
- $D$ is a (non-empty, finite) domain of discourse;
- $V$ is a value-assignment (see below).

We define a *value-assignment*, $V$, as follows:

- for each constant $d \in D$, $V(d) = d$,
- for each variable $X$, $V(X) = d$, where $d \in D$ is a constant,
- for each $k$-ary predicate, $P$, $V(P) \subseteq D^k \times W$ is a set of $(k+1)$-tuples, $(d_1, \ldots, d_k, w)$, with $w \in W$, which are assigned the value $\mathtt{T}$ for the predicate $P$.

$V$ assigns a truth value to each atom $P(t_1, \ldots, t_k)$, not preceded by the operator $\Box$, at each world $w \in W$, as follows: $V(P(t_1, \ldots, t_k), w) = \mathtt{T}$ iff $(V(t_1), \ldots, V(t_k), w) \in V(P)$; V assigns a truth value to every formula as follows:

- $V((L, w) = \mathtt{T}$ iff $\neg(V(L, w)) = \mathtt{T}$ (the CWA applies)
- $V(L \wedge K, w) = \mathtt{T}$ iff $V(L, w) = \mathtt{T}$ and $V(K, w) = \mathtt{T}$
- $V(K \leftarrow L, w) = \mathtt{T}$ iff $V(K, w) = \mathtt{T}$ whenever $V(L, w) = \mathtt{T}$
- $V(\Box L, w) = \mathtt{T}$ iff for all $z \in W$ such that $(w, z) \in AR$, $V(L, z) = \mathtt{T}$

where $L$ and K are formulae.

We observe that $V$ gives the truth value of a formula at a particular point $w \in W$, and we write $\mathcal{M} \models_w L$, which is read as "for the **S5**-model $\mathcal{M}$ formula $L$ is true in world $w$".

We restrict our attention to the truth value of a formula, $L$, at the particular point $wa \in W$ which we use to denote the *actual* world. For the sake of simplicity, we will write $\mathcal{M} \models L$ as shorthand for $M \models_{wa} L$. (This is an abuse of the standard notation, where $\mathcal{M} \models L$ means that $L$ is $\mathtt{T}$ at all worlds in the universe.)

Given an **S5**-model, $\mathcal{M}(W, AR, D, V)$, and a finite set of formulae, $\Xi$, we say $\mathcal{M}$ is a $w$-model of $\Xi$ if $M \models_w L$ for all $L \in \Xi$. From now on, given a set of formulae, $\Xi$, which we write in the context of the actual world $wa \in W$, we concern ourselves with a $wa$-model whose value-assignment is the set of all ground atoms which are a logical consequence of $\Xi$. We will refer to this as the *intended model* for $\Xi$.

**Example 3.1** *Consider the set of formulae* $\{P(X) \leftarrow \Box Q(X), \Box Q(a)\}$ *and define* $W = \{wa, wn\}$, *and* $AR = \{(wa, wa), (wa, wn), (wn, wn), (wn, wa)\}$. *(wn represents a world which is distinct from the actual world wa.) In our intended model, the valuation,* $V$, *is given by*

$$V(Q) = \{Q(a, wa), Q(a, wn)\} = \{Q(a), \Box Q(a)\} \textit{ and } V(P) = \{P(a, wa)\} = \{P(a)\},$$

*in accordance with an intuitive interpretation of the logical consequences of the formulae.*

*(We observe that* $M \not\models_{wn} P(X) \leftarrow \Box Q(X)$, *where* $V$ *is as defined in the preceding paragraph, illustrating that our intended model is not a wn-model.)*

**Definition 3.1** *Let $\Xi$ be a finite set of formulae in our modal logic and let $K$ be the set of constants in $\Xi$. We define $\Xi^K$ to be the finite set of all ground atoms that can be constructed using the modal operator $\Box$ and the predicates and constants appearing in $\Xi$.*

We note that the value-assignment in the intended model for $\Xi$ can be thought of as a subset of $\Xi^K$. (That is, if $V(P(t_1, \ldots, t_k), w) = \mathtt{T}$ for all $w$ such that $(wa, w) \in AR$, we identify this with $\Box P(t_1, \ldots, t_k) \in \Xi^K$; and if $V(P(t_1, \ldots, t_k), wa) = \mathtt{T}$ we identify this with $P(t_1, \ldots, t_k) \in \Xi^K$ as in the preceding example.)

**Definition 3.2** *Let $\Xi$ be a finite set of formulae, and let $W$ be a finite set of worlds. Given a value-assignment, $I$, the consequence operator, $IC : \mathcal{P}(\Xi^K) \rightarrow \mathcal{P}(\Xi^K)$, where $\mathcal{P}(\Xi^K)$ denotes the power set of $\Xi^K$, is defined by*

$$IC(I) = \{L : L \leftarrow L_1 \wedge \ldots \wedge L_k \textit{ is a ground instance of a formula in } \Xi,\ \{L_1, \ldots, L_k\} \subseteq I\}$$

We observe informally that $IC$ extends a given value-assignment, $I$, for the predicates in $\Xi$ to include facts that can be inferred from $I$ and the formulae in $\Xi$.

**Definition 3.3** *We define $I_0 = \emptyset$ and $I_n = IC(I_{n-1})$.*

Appendix I presents formal results regarding the existence and uniqueness of a *fixpoint*, $I_N$, of the operator $IC$. $I_N$ represents the set of logical consequences of the set of formulae at the actual world.

**Example 3.2** *Using our previous example we have*

$$I_0 = \emptyset,\ I_1 = \{\Box Q(a)\},\ I_2 = \{Q(a), \Box Q(a), P(a)\} = I_3 = \cdots.$$

*We observe that:*

- *$Q(a)$ is entered into $I_2$ since $\Box L \leftarrow L$ is an axiom in* **S5***;*
- *$I_2$ is a fixpoint.*

### 3.3 From Modal Logic to First Order Logic

We will want to render our finite set of formulae, $\Xi$, in first order logic so as to use logic programming tools such as Prolog or Datalog. The following theorem states that such a translation is feasible.

**Theorem 3.1** *Given a finite set of formulae, $\Xi$, and the intended model $\mathcal{M}(W, AR, D, V)$, where $W$ is finite, $\Xi$ can be transformed in polynomial time into $\Pi$, a set of formulae in classical first-order predicate logic.*

**Proof:** Following Levene and Loizou [12], we describe a transformation from a finite set of formulae $\Xi$ of our modal logic to a set of formulae $\Pi$ of first-order predicate logic.

Given the set of worlds $WA = \{wa, w_1, \ldots, w_p\} \subseteq W$, where $(wa, w_i) \in AR$ for all $i$, $1 \leqslant i \leqslant p$, the transformation is defined as follows.

- We introduce constants $wa$ and $w_i$, $1 \leqslant i \leqslant p$, into $\Pi$ to represent the worlds in $WA$. We assume that we can name these constants in such a way that they do not already appear in $\Xi$.
- Replace the head of each rule (including atomic formulae which can be regarded as rules with no bodies) in $\Xi$ as follows:

– if the rule has the form $P(t_1, \ldots, t_k) \leftarrow L$, where $L$ is the body of the rule, replace it with the rule

$$P(t_1, \ldots, t_k, wa) \leftarrow L;$$

– if the rule has the form $\Box P(t_1, \ldots, t_k) \leftarrow L$, where $L$ is the body of the rule, replace it with the rules

$$\begin{gathered} P(t_1, \ldots, t_k, wa) \leftarrow L, \\ P(t_1, \ldots, t_k, w_1) \leftarrow L, \\ \vdots \\ P(t_1, \ldots, t_k, w_p) \leftarrow L. \end{gathered}$$

- Replace each instance of $P(t_1, \ldots, t_k)$ in the body of the resulting rules by

$$P(t_1, \ldots, t_k, wa),$$

and each instance of $\Box P(t_1, \ldots, t_k)$ by

$$P(t_1, \ldots, t_k, wa) \wedge P(t_1, \ldots, t_k, w_1) \wedge \ldots \wedge P(t_1, \ldots, t_k, w_p).$$

It is easy to verify that this transformation requires polynomial time in the size of $\Xi$ the size of $WA$ and the number of predicates in $\Xi$. ■

Our transformation preserves the validity of theorems in $\Xi$ with respect to the intended model $\mathcal{M}$. This is a special case of a result due to Morgan [18]. Specifically, if in our model $\mathcal{M} \models L(...)$, where $L(...)$ is a fact, then $L(..., wa)$ is a logical consequence of the formulae in $\Xi$, and if $M \models \Box L(...)$, then $L(..., wa)$ and $L(..., w_i)$ for all $i$, $1 \leqslant i \leqslant p$, are logical consequences of the formulae in $\Pi$.

We now simplify the transformation into first order logic by reducing the size of our universe of worlds. We note that to make meaningful use of the modal operator $\Box$, we require that we can distinguish between the semantics of $L$ and $\Box L$ at the world $wa \in W$.

**Theorem 3.2** *For the model $\mathcal{M}(W, AR, D, V)$, $|W| > 1$, where $|W|$ is the cardinality of the set $W$, is a necessary and sufficient condition for the formulae $L$ and $\Box L$ to have different semantics at the world $wa \in W$.*

**Proof:** (Sufficiency) Let $W = \{wa, wn\}$ and $AR = \{(wa, wa), (wn, wn), (wa, wn), (wn, wa)\}$. The model $\mathcal{M}$ provides a semantics to the formulae $L$ and $\Box L$ as follows:

- $\mathcal{M} \models_{wa} L$ if, and only if, $V(L, wa) = \mathtt{T}$;
- $\mathcal{M} \models_{wa} \Box L$ if, and only if, $V(L, wa) = \mathtt{T}$ and $V(L, wn) = \mathtt{T}$.

Thus we have $L \not\equiv L$.

(Necessity) Consider the model $\mathcal{M}_1(W1, AR_1, D, V)$, where $W_1 = \{wa\}$, $AR_1 = \{(wa, wa)\}$, that is, $|W| = 1$. Then we have in particular $M \models_{wa} L$ if, and only if, $V(L, wa) = \mathtt{T}$; $\mathcal{M} \models_{wa} \Box L$ if, and only if, $V(L, wa) = \mathtt{T}$. Thus, for $|W| = 1$, $L \equiv \Box L$. This establishes the result. ■

**Definition 3.4** *Given a finite set of formulae, $\Xi$, and a model $\mathcal{M}(W, AR, D, V)$ for $\Xi$, a* query *is an expression of the form $\leftarrow L_1 \wedge \ldots \wedge L_k$. Such a query is also denoted by $?L_1 \wedge \ldots \wedge L_k$.*

*The* answer *to a query is a subset of $\Xi$ which represents all ground instances of the query expression which evaluate to T in the intended model. (If the answer is the empty set the query is said to* fail.*)*

For a more detailed analysis of queries in first order logic, and the way in which logic programs compute the answer the reader is referred to [14].

**Example 3.3** *We transform the formulae*

$$\{P(X) \leftarrow \Box Q(X), \Box Q(a)\}$$

*into*

$$\{P(X, wa) \leftarrow Q(X, wa) \wedge Q(X, wn), Q(a, wa), Q(a, wn)\}.$$

*The query* $?\Box P(X)$*, for example, is replaced by* $?P(X, wa) \wedge P(X, wn)$ *which fails as a first order logic query in our transformed logic.*

**Definition 3.5** *We define* $\Delta$*, a deductive database, to be a finite set of formulae describing an access control mechanism. The set* $\Delta$ *is the union of a set* $\Delta_E$ *of ground positive facts and a set* $\Delta_I$ *of inference rules [13].*

# 4 The Logical Language

In this section we present the predicates and inference rules of our logic, and a number of queries posed to our deductive database $\Delta = \Delta_E \cup \Delta_I$. For paedagogical reasons we will present the predicates and inference rules of our logic as a classical first-order predicate logic in the first instance, to which we will later add the modal operator $\Box$ where appropriate.

## 4.1 Predicates

In Table 2 we introduce the predicates of our language, with reference to the formal model of Section 2. The column labelled "Semantics" states necessary and sufficient conditions for the predicate to be true.

| Predicate Name | Semantics |
|---|---|
| `request`$(q)$ | $q \in Q$ |
| `needs`$(x, y)$ | $(x, y) \in N_R$ (Definition 2.1) |
| `have`$(x)$ | $x \in M$ |
| `parent_of`$(o_1, o_2)$ | $(o_1, o_2) \in E$ (Definition 2.5) |
| `searches`$((o_1, s, r), o_2)$ | $o_2 \in O$ is used when resolving some other object's name $o_1 \in O$, $o_1 \neq o_2$, in the course of executing the request $(o_1, s, r)$ |
| `effective`$(q)$ | $q \in Q,\ c_q \subseteq M$ |
| `requires`$(q, r)$ | $q \in Q,\ r \in c_q$ |
| `lacks`$(q, r)$ | $r \in c_q,\ r \notin M$ |

Table 2: Predicates

## 4.2 Inference Rules

We define a rule to associate the `needs` and `request` predicates.

$$\texttt{needs}(x, x) \leftarrow \texttt{request}(x) \tag{1}$$

The `needs` predicate is transitively closed under the following rule.

$$\texttt{needs}(x, z) \leftarrow \texttt{needs}(x, y) \wedge \texttt{needs}(y, z) \tag{2}$$

The following rules infer the set of object names searched when resolving an object's name, $o_1 \in O$.

$$\texttt{searches}((o_1, s, r), o2) \leftarrow \texttt{parent\_of}(o_2, o_1) \quad (3)$$
$$\texttt{searches}((o_1, s, r), o_3) \leftarrow \texttt{searches}((o_1, s, r), o_2) \wedge \texttt{parent\_of}(o_3, o_2) \quad (4)$$
$$\texttt{needs}((o_1, s, r), (o_2, s, search)) \leftarrow \texttt{searches}((o_1, s, r), o_2), \text{ where } search \in R \quad (5)$$

We define the `requires` predicate as follows.

$$\texttt{requires}((o_1, s_1, s_1), (o_2, s_2, r_2)) \leftarrow \texttt{needs}((o_1, s_1, r_1), (o_2, s_2, r_2)) \quad (6)$$
$$\texttt{requires}((o_1, s_1, r_1), (o_2, s_1, search)) \leftarrow \texttt{searches}((o_1, s_1, r_1), o_2) \quad (7)$$

We define the `effective` predicate as follows.

$$\texttt{effective}(q) \leftarrow \texttt{request}(q) \wedge \neg(\texttt{needs}(q, x) \wedge \neg\texttt{have}(x)) \quad (8)$$

Finally, we define the `lacks` predicate as follows.

$$\texttt{lacks}(q_1, q_2) \leftarrow \texttt{requires}(q_1, q_2) \wedge \neg(\texttt{have}(q_2)) \quad (9)$$

In other words, (8) states that a request is effective if there are no access rights that the request needs which are not granted. The CWA states that any facts which are not specifically stated to be true are assumed to be false, and is adopted in logic programming systems such as Prolog, where a negative ground literal $\neg x$ is inferred whenever a positive ground literal $x$ cannot be deduced from the given facts in $\Delta_E$. In general, the CWA can only be applied consistently to databases that contain ground positive literals and Horn clauses [14]. Further, negations in the body of an inference rule may lead to an inconsistent theory when the CWA is employed, because we may infer that a fact is true under the CWA, and then introduce a new fact that is the negation of the inferred fact [26]. The inference rules given in (8) and (9) differ from the earlier inference rules in that a negative literal appears in the body of each rule. In the specific case of our logic this does not lead to inconsistency because

- the negative literals, ¬`have` and `needs`, are neither expressed as facts nor explicitly inferred by any of the inference rules;
- neither `effective` nor ¬`effective` are ever expressed as facts.

In some operating systems, one type of access right may imply another, for example, "*write*" $\in R$ may implicitly grant "*append*" $\in R$. Such cases can be accommodated by including rules like (10) below. Table 6 on page 15 shows the inference rules for the Windows NT operating system.

$$\texttt{have}((n, s, append)) \leftarrow \texttt{have}((n, s, write)) \quad (10)$$

In Table 3 we summarise the predicates and inference rules of our language.

### 4.3 Queries

We now define a number of standard queries which we used in our experiments and which operate against a database, $\Delta$, of facts and inference rules of our logic. We will write the definitions of our queries in the syntax of Prolog; thus we use the symbols $o$, $s$ and $r$ to stand for constants in the sets $O$, $S$ and $R$, respectively, and we use the symbols $O$, $S$ and $R$ to refer to variables over the aforesaid sets. Hereafter we use the placeholder, $W$, to indicate modality in a query, on the understanding that $wa$ or $wn$, as appropriate, would be substituted for $W$ at query time. The queries are defined below with respect to the current state, $M$, of the access control mechanism, ACM.

| | |
|---|---|
| needs | $\texttt{needs}(x, x) \leftarrow \texttt{request}(x)$<br>$\texttt{needs}(x, z) \leftarrow \texttt{needs}(x, y) \wedge \texttt{needs}(y, z)$<br>$\texttt{needs}((o_1, s, r), (o_2, s, search)) \leftarrow \texttt{searches}((o_1, s, r), o_2)$ |
| searches | $\texttt{searches}((o_1, s, r), o2) \leftarrow \texttt{parent\_of}(o_2, o_1)$<br>$\texttt{searches}((o_1, s, r), o_3) \leftarrow \texttt{searches}((o_1, s, r), o_2) \wedge \texttt{parent\_of}(o_3, o_2)$ |
| requires | $\texttt{requires}((o_1, s_1, r_1), (o_2, s_2, r_2)) \leftarrow \texttt{needs}((o_1, s_1, r_1), (o_2, s_2, r_2))$<br>$\texttt{requires}((o_1, s_1, r_1), (o_2, s_1, search)) \leftarrow \texttt{searches}((o_1, s_1, r_1), o_2)$ |
| effective | $\texttt{effective}(q) \leftarrow \texttt{request}(q) \wedge \neg(\texttt{needs}(q, x) \wedge \neg\texttt{have}(x))$ |
| lacks | $\texttt{lacks}(q_1, q_2) \leftarrow \texttt{requires}(q_1, q_2) \wedge \neg(\texttt{have}(q_2)$ |

Table 3: Inference Rules

To obtain the set of requests that would necessarily complete successfully (in other words, all possible triples that a request may need are granted):

$$?\texttt{effective}((O, S, R), wa).$$

To obtain the set of requests that would not necessarily complete successfully:

$$?\texttt{effective}((O, S, R), wn).$$

To obtain the set of triples that must be granted in $M$ in order for a given request, $(o, s, r)$, to complete successfully:

$$?\texttt{requires}((o, s, r), (O, S, R), W).$$

To determine the set of requests which require a given triple:

$$?\texttt{request}((O, S, R), W), \texttt{requires}((o, s, r), (O, S, R), W).$$

To determine the set of triples whose absence is preventing the successful completion of a given request, $(o, s, r)$,

$$?\texttt{lacks}((o, s, r), (O, S, R), W).$$

To determine the set of requests that would fail were a given triple, $(o, s, r)$, revoked:

$$?\texttt{request}((O, S, R), wn), \texttt{effective}((O, S, R), wa), \texttt{requires}((O, S, R), (o, s, r), wn).$$

We note that it is not possible to infer facts concerning the revocation of a triple which is possibly required by a request that possibly completes successfully; there is no way to determine whether there are circumstances in which the request can complete successfully without attempting to invoke that triple.

## 4.4 Prolog Implementation

We discuss the Prolog implementation of our first order language using an example. Assume that the following segment of code forms part of a UNIX shell script stored in a file named $/test1/test.sh$. By construction, this script will always attempt to read the file $/test1/message$ (line 1), will possibly attempt to write into the file $/test1/gt$ (line 3), and will possibly attempt to write into the file $/test1/le$ (line 4). In other words, a request, $q \in Q$, where $q$ is a triple granting the right to execute the file $/test1/test.sh$, necessarily requires triples to execute the cat program and to read from the file $/test1/message$, possibly requires a triple to write into the file $/test1/gt$

| | |
|---|---|
| `/bin/cat` $/test1/message$ | line 1 |
| `if` [ "\$1" -gt "0" ] | line 2 |
| `then echo` "\$1" >> $/test1/gt$ | line 3 |
| `else echo` "\$1" >> $/test1/le$ | line 4 |
| `fi` | line 5 |

and possibly requires a triple to write into the file $/test1/le$. This can be expressed in terms of our first order logic predicates and the necessity operator as follows.

$$\Box\texttt{request}((/test1/test.sh, S, execute)).$$
$$\Box\texttt{needs}((/test1/test.sh, S, execute), (/bin/cat, S, execute)).$$
$$\Box\texttt{needs}((/test1/test.sh, S, execute), (/test1/message, S, read)).$$
$$\texttt{needs}((/test1/test.sh, S, execute), (/test1/gt, S, write)).$$
$$\texttt{needs}((/test1/test.sh, S, execute), (/test1/le, S, write)).$$

Defining a Prolog rule of the form $P(t_1, \ldots, t_n, wa) :- P(t_1, \ldots, t_n, wn)$ for each predicate, $P$, is an alternative, and more economical, way of entering the facts $P(t_1, \ldots, t_n, wn)$ and $P(t_1, \ldots, t_n, wa)$ into the extensional database. This means, for example, that we could define the rule

$$\texttt{have}(Q, wa) :- \texttt{have}(Q, wn),$$

and subsequently need only enter $\texttt{have}(x, wn)$ and $\texttt{have}(y, wn)$, to enable us to infer $\texttt{have}(x, wa)$ and $\texttt{have}(y, wa)$. That is, in general we can represent $k$ necessary (modal) facts for a given predicate using $(k+1)$ instead of $2k$ first order facts. The modal facts above can then be expressed in Prolog as follows.

$$\texttt{request}((O, S, R), wa) :- \texttt{request}((O, S, R), wn).$$
$$\texttt{request}((/test1/test.sh, S, execute, wn)).$$
$$\texttt{needs}((O1, S1, R1), (O2, S2, R2), wa) :- \texttt{needs}((O1, S1, R1), (O2, S2, R2), wn).$$
$$\texttt{needs}((/test1/test.sh, S, execute), (/bin/cat, S, execute), wn).$$
$$\texttt{needs}((/test1/test.sh, S, execute), (/test1/message, S, read), wn).$$
$$\texttt{needs}((/test1/test.sh, S, execute), (/test1/gt, S, write), wa).$$
$$\texttt{needs}((/test1/test.sh, S, execute), (/test1/le, S, write), wa).$$

# 5 Experimentation

We performed some experiments to demonstrate that our logic is practical to implement and that it can be applied to a real-world operating system. As a result of these experiments, we introduced a number of important features to our logic.

Our experimentation made use of the UNIX operating system produced by the Santa Cruz Operation Ltd (SCO) and the Windows NT operating system produced by Microsoft [5, 25]. Both operating systems are designed to meet the security requirements of level C2 of the Trusted Computer System Evaluation Criteria [2], as a result of which they both feature security audit trails from which we obtained many of the facts for our deductive database.

Our approach was to obtain a set of ground facts from each operating system using an assortment of imperative programs, and to present these facts to Prolog. For these experiments we did not attempt to describe in Prolog the specific implementation details of either operating system, such as groups or the syntax and semantics of access control lists [22].

The first task was to produce an implementation of our logic in a suitable programming language. We chose to use Prolog for our experiments because it is well known, it is sufficient for our purposes and because interpreters for Prolog are available from a variety of sources. The interpreter that we used was the MS-DOS version of the SB-Prolog System, version 3.1, from the University of Arizona [6]. This is a rather modest system, but it proved to be adequate for the purposes of our experiments.

A number of additional predicates were introduced in the Prolog encoding for implementation reasons, for example, in order to avoid looping when the Prolog interpreter encountered recursive inference rules. Overall we found that it was straightforward to implement our logic in Prolog.

Our first experiment was performed on an instance of the SCO UNIX operating system, comprising 25 users and 22 user groups. This experiment had three objectives. First, to establish the practicality of dealing with the number of facts required to describe a real-world operating system. (In general, we suspect that automated construction of a deductive database is essential to the practical application of our approach.) Second, to discover a way of obtaining facts directly from a UNIX file system. Third, to apply our logic to some real-world computations.

Initially, we used a UNIX script program to output the set of `parent_of` facts for the SCO UNIX file system, and a C program to output the entire set of have facts for a SCO UNIX raw disk device. This resulted in 7,555 `parent_of` facts and 222,802 `have` facts, which was beyond the capacity of the SB-Prolog interpreter. We therefore discarded any facts pertaining to subjects and objects other than those that were actually used in the course of our experiment.

We performed a number of controlled activities on the SCO UNIX operating system, for which the corresponding set of `needs`and request facts were obtained from the security audit trail. From these we obtained the set of subject and object names, and generated a set of `parent_of`facts. Only seven subject names appeared in the facts. We next obtained the set of `have`facts for the objects in question using a C program whose input consisted of a list of object names and whose output was the set of triples granted for each object. Any `have`facts that referred to subjects other than the seven above were deleted. (Our database did not contain any `request`facts for these subjects, and we were inferring `need`ed triples with reference to specific requests.) Further, all `have`facts associated with the UNIX root account were deleted, and replaced instead by a single rule saying that the root subject has every possible type of access right to every object.

The number of facts obtained is shown in Table 4 below. We observe that the number of possible `have`facts increases polynomially as users (and, similarly, objects or access rights) are added to the operating system. The sets of facts plus the Prolog inference rules formed our

| *fact* | *number* |
|---|---:|
| `parent_of` | 282 |
| `request` | 14 |
| `needs` | 2821 |
| `have` | 1197 |

Table 4: Number of Facts in Deductive Database $\Delta$

deductive database. We experienced some minor problems when we first ran our queries, such as a recursive `needs` fact relating the object "/" to itself which caused the Prolog interpreter to loop. However, these problems were all easy to identify and straightforward to correct.

Some of the answers to the queries were unexpected in that they showed that some of our requests required triples which we had not expected, such as triples for keyboard and terminal character mapping files used by the UNIX shell command interpreter. This information might benefit a system administrator (and his or her users) who is contemplating revoking these triples and is unaware of their importance. Without this information, essential access rights might be revoked, leading to process failures.

The results of our first experiment demonstrated that it is possible to implement our modal logic using a logic programming language, and that our logic succeeds in answering some questions of practical importance.

In order to investigate the generality of our logic, and the difficulty of adapting it to a different type of operating system, we conducted a second experiment using Windows NT [21]. In this experiment we constructed a database describing the set of access rights required to run the commercial "Microsoft Word" word processing program against a set of files in an NTFS file system.

Most of the inference rules required for describing Windows NT could be copied from those describing SCO UNIX; those that differed are discussed below. There is no "`root`" user in Windows NT, so the inference rule referring to the `root` user was omitted. Table 5 indicates the abbreviations used for access rights supported by access control lists in Windows NT. Table 6 shows the inference rules describing relationships that hold between access rights in an NTFS file system [5, 25, 17]. These rules enable us to reduce the size of our database by omitting large numbers of have facts and inferring them instead from more "permissive" access rights.

| | |
|---|---|
| $x$ | `execute` |
| $r$ | `read` |
| $c$ | `change` |
| $d$ | `delete` |
| $w$ | `write` |
| $p$ | `change properties` |
| $o$ | `take ownership` |
| $a$ | `all` |

Table 5: NT Access Rights

| |
|---|
| $\texttt{have}((O,S,x),W) :- \texttt{have}((O,S,r),W).$ |
| $\texttt{have}((O,S,r),W) :- \texttt{have}((O,S,c),W).$ |
| $\texttt{have}((O,S,d),W) :- \texttt{have}((O,S,c),W).$ |
| $\texttt{have}((O,S,w),W) :- \texttt{have}((O,S,c),W).$ |
| $\texttt{have}((O,S,p),W) :- \texttt{have}((O,S,a),W).$ |
| $\texttt{have}((O,S,o),W) :- \texttt{have}((O,S,a),W).$ |
| $\texttt{have}((O,S,c),W) :- \texttt{have}((O,S,a),W).$ |

Table 6: Inference Rules for NT Access Rights

We switched on audit trail recording, then launched and ran the word processor. As before, we obtained a set of `request` and `needs` facts by analysis of the security audit trail. There was insufficient information in the security audit trail to clearly distinguish a request fact from a `needs` fact; consequently we had to add request facts to our database manually. With the SCO UNIX security audit trail we had been able to distinguish between request facts and `needs` facts by reconstructing the process tree, rooted at the login shell. This is a problem that could be easily overcome by adding further information to the security audit trail. Again our approach was to obtain a set of ground `have` and `parent_of` facts directly from the operating system, without modelling in Prolog the native Windows NT mechanisms such as object owner and subject groups. This is significant because the mechanisms used in Windows NT allow for the explicit declaration of denied access rights for both subjects and subject groups. The algorithm used by Windows NT for resolving any inconsistencies between permitted and denied access rights is highly specialised and imperative in nature. The restrictions placed on our logic, which ensure its efficiency, exclude negative facts, so that in fact we cannot describe the NT mechanisms directly in our logic. This presents no problem when we deal with ground `have` facts.

We used the `perms` program from the "Windows NT Resource Kit" to obtain the set of `have` facts and `parent_of` facts. We then omitted any triples that could be inferred from the above rules, thereby reducing the number of `have` facts from 5072 to 2820.

Initially, our queries indicated that a significant number of triples were missing in respect of temporary files created by the Word program. The Word program deleted these files when it terminated, so the files did not exist at the time that we obtained our `have` facts. All of these files had very distinctive names, and we were able to account for them by introducing `have` facts that matched the name syntax of the temporary file names. Following this, our standard queries returned the expected results.

Overall, the effort required in order to adapt our logic to the Windows NT operating system was not substantial. Most of the effort was spent implementing simple programs for obtaining facts from Windows NT.

We recognise that further experimentation is required in order to assess the scalability of our approach. We have already conducted experiments where file definitions are used as Prolog facts and infer `have` and `parent_of` facts from them, in an attempt to reduce the number of ground facts in the deductive database. A simple example is included in the next section, and further examples can be found in [4].

We also recognise that our techniques to generate `needs` facts are ad hoc and do not guarantee that all dependencies between access rights have been identified. Further work needs to be undertaken to establish whether, in fact, such a guarantee can be delivered, and, if so, how that might be achieved.

# 6 Specification and verification of security policy

A comprehensive review of specification and verification of security policies is found in [20], from which we adopt the following definition of an Access Control Policy (ACP). An ACP specifies those states of ACM that preserve certain desirable properties of information confidentiality, integrity and availability for the operating system in question.

In the ensuing definition the notion of *consistency* is employed. In simple terms, the state, $M$, of an ACM is consistent with a given ACP if it satisfies the requirements of that policy. For a formal treatment of this concept, the reader is referred to [4].

**Definition 6.1** *An operating system is in a* secure *state* $k = (F, A, M)$ *iff* $M$ *is consistent with the ACP given for that operating system. In such a case we also say that* $M$ *is a* secure *state of the ACM.*

We have seen that a deductive database implemented in Prolog is capable of describing the ACM of a real-world operating system, and of reasoning about the effects of an ACM on the computations performed by the operating system.

The overwhelming number of subjects, objects and types of access right in a real-world operating system means that a system administrator will not always fully appreciate the consequences of changing an ACM. Our results provide a basis for developing automated tools that are capable of describing, and reasoning about, such large numbers of facts, and hence improve the understanding of the implications of changing an ACM, and hence reducing the possibility of configuring an ACM in such a way as to compromise the corresponding ACP.

We now explore whether Prolog might provide a suitable system for writing an ACP that could be used to determine whether an ACM described in our deductive database is in a secure state.

We demonstrate this via a simple example. Although various specification schemes for ACPs exist in the literature [20], we use Prolog clauses to specify our ACP and choose the UNIX operating system to provide an ACM since the semantics of its file permissions align closely to Prolog's negation-as-failure rule.

From our example we hope to gain some insight into the following issues.

- How do the restrictions of Prolog limit its suitability for writing an ACP?
- How easy is it to write an abstract ACP in terms of real-world objects and operations, and to map the ACP onto a real-world access control mechanism?
- How readily can the specification of security given in the ACP be used to verify whether a given state of an access control mechanism is a secure state?

Our example concerns a university computer science department in which the following activities must be supported.

- A lecturer can set and mark examination papers in a variety of subjects. Papers and marks are written to files.
- A student can read examination papers, but not files containing marks.
- A student can submit an examination script (as a file) for marking if the student studies the course on which the paper is set.

The following approach is adopted.

- We write an abstract ACP in Prolog. It is intended that this should use, as far as possible, a natural (language) interpretation of the requirements of the ACP. The specification is abstract in the sense that the ACP is written without reference to the operating system which is to provide the ACM.
- We describe the ACM implementation using Prolog to simulate a set of UNIX files, users and groups.
- We map requirements of the ACP onto requests of the form (file, user, access right).

A fuller account of this example and others can be found in [4]. We adopt the usual Prolog syntax where variable names start with an upper case letter, constant names with a lower case letter and lists are comma-separated and enclosed in square brackets. However, we omit the single quote delimiters for arbitrary constants in the interests of readability.

The following extract forms a typical part of our ACP specification. The `person` predicate is defined by a list of facts specifying the users of the proposed file system, and takes three arguments - the name and status of the person, and a list of the courses with which the person is involved. (The `member` predicate is true if and only if the first argument is a member of the second argument, which must be a list.)

$$\begin{aligned}\texttt{can}(Course, Name, sit) :- \ & \texttt{person}(Name, student, Courses),\\ & \texttt{member}(Course, Courses).\end{aligned}$$

The intended meaning of the above clause is that a person can sit an examination paper if (s)he is a student and studies the course for which the paper was set. The UNIX implementation is written in Prolog by expressing the UNIX accounts and files in the system as facts using the predicates, `user_def` and `file_def`. The former takes two arguments, the account name and a list of groups to which that account belongs. The latter takes four arguments, the file name, its owner, its group owner and its permissions. The file permissions are written as a list in order to facilitate processing. Typical examples are shown below. (We assume that there is only one course, and that all material relating to that course is held in a sub-directory, */cs*, of the root directory. The directories */papers*, */scripts* and */marks* are sub-directories of the */cs* directory.)

$$\begin{aligned}&\texttt{user\_def}(jc, [students]).\\ &\texttt{file\_def}(/cs/papers, root, staff, [d, r, w, x, r, w, x, r, -, x]).\end{aligned}$$

We infer the `have` and `needs` facts directly from the file and user definitions. (It was not felt necessary to explicitly create a deductive database of triples or to use our modal logic operator $\Box$ for this example.)

We map the ACP onto the UNIX implementation using the predicate `ait_map`. The approach that we adopt is to equate each action in the ACP with a set (represented as a list) of UNIX access rights. The extract below illustrates the code used.

$$\texttt{ait\_map}((cs, Name, mark), [(/cs/scripts, Acc, r), (File, Acc, r), (/cs/marks, Acc, w)]) :-$$
$$\texttt{ais\_map}(Name, Acc), can(cs, Name, mark),$$
$$\texttt{parent\_dir}(/cs/scripts, File).$$

In this example, marking a script maps to requests to read the */cs/scripts* directory, to read a file from that directory, and to write to the */cs/marks* directory. The `ais_map` predicate maps user names in the ACP onto UNIX accounts in the implementation. (The predicate names `ais_map` and `ait_map` are contractions of abstract-implementation-subject-map and abstract-implementation-triple-map, respectively.) The `parent_dir` predicate is true if the first argument is the parent directory of the second argument.

The above framework enables us to pose the following questions as Prolog queries:

Q1 Which UNIX access rights need to be granted to implement the ACP? That is, which access rights need to be granted so that every request that can be inferred from the `ait_map` predicate completes successfully? This query was constructed in order to provide an easy means of defining the two following queries.

Q2 Which of the above access rights are not granted in a given state of the UNIX implementation?

Q3 Which access rights granted in a given state of the UNIX implementation are not authorised by the ACP? That is, is our UNIX implementation in a secure state with respect to our ACP?

The Prolog interpreter returned "no" in response to Q2 (that is, the implementation granted all the access rights required by the ACP) as we expected. However, the interpreter did return non-empty responses to Q3. These were of two types.

- Access rights that allowed a student to read the script of another student. Presumably this would be undesirable, but it should be noted that the ACP does not preclude this possibility. That is, the ACP has been inadequately specified.
- Access rights that allowed a user read access to a directory. These instances are examples of the fact that read access to a directory in UNIX is equivalent to list access. It does not actually convey any rights to access files in the directory. In particular, it is quite common to have execute access, but not read access, to a directory. This enables users to read (or write to) any files in the directory (that they know the name of) and to which they have read (or write) access, but prevents them from listing the contents of the directory [19]]. Thus our simple experiment raised the question "to which directories is it appropriate for users to have both read and execute access?" For example, it would be reasonable in the light of our experiment, to restrict access to the /papers directory to execute only, and in "real life" for students to be given the path and filename for the paper they were required to sit.

In summary, we generate a list of requests that our implementation must provide in order to support all the operations in the ACP, which in turn gives rise to a set of access rights that are required for those requests to complete successfully. We can now analyse any access rights granted by the ACM that are not permitted by the ACP specification, and assess whether they arise as a result of incorrectly, or partially, specifying the ACP, incorrectly implementing the ACM, or indeed whether they are an unavoidable consequence of the shortcomings of the ACM mechanism provided.

This example demonstrates that it is possible to specify an ACP using Prolog and, from our experience, indicates that doing so can draw attention to possible security weaknesses in the state of an ACM intended to implement that ACP. Prolog seems to provide a suitable means of specifying an ACP, and of describing an ACM in the UNIX operating system, given the similar restrictions of Prolog's negation as failure rule, and UNIX's denial of an access right if it is not explicitly granted.

# 7 Conclusions

We have described a modal logic that allows us to reason about the effects of an ACM on the computations performed by an operating system. Experimentation has demonstrated that such a logic is practical to implement using existing tools and, moreover, for a reasonably large number of facts. These results establish an unexpected foundation for improving the effectiveness of access control mechanisms, and thereby for improving operating system security, through the use of logic programming and deductive database technology.

There are numerous opportunities for further research. We believe that the existing formal foundations are sufficient for practical experimentation to lead the way. The formalism will be extended where experimentation shows it to be necessary. Tools to support experimentation are readily available in the form of logic programming systems, and in any case we believe that more experimentation is desirable to establish the benefits and limitations of the logic as currently formulated. We therefore intend to conduct experiments to investigate the use of our approach on a variety of real-world operating systems. We are now considering extensions to our model that would enable us to examine:

- distributed systems, where we would like to incorporate some of the ideas in the calculus for access control in distributed systems developed by Abadi *et al.* [1];
- heterogeneous systems sharing a common ACP;
- generic system administration activity;
- role-based access control;
- the management of sensitive combinations of access rights;
- and explicit modelling of implementation details including object owner, groups and access control lists.

Our experiment in Section 6 also raises several questions. Can we use our techniques on other platforms? For example, could we map the same ACP onto the ACM of a UNIX system and the ACM of Windows NT? Prolog, which does not support explicit negation, presumably prevents us from specifying some types of ACP, and different formulations will presumably be needed to cope with ACMs which support explicit denial of access rights. We anticipate that logic programming systems such as Datalog and applications of default logic may provide some solutions to these issues [3, 10]. We anticipate, however, that much may be gained from a better understanding of how much can be achieved using a Prolog-based implementation.

We envisage that stronger concepts of object-orientation may also be required, together with some form of calculus operating upon object types, in order to facilitate the construction of mappings between an ACP and a deductive database. We are currently considering such extensions to the logic.

**Acknowledgements** The work of Jason Crampton has been supported by an EPSRC award. The authors would like to thank the referees for their constructive comments.

## References


[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[2] D.E. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Volume I, Mitre Corporation, March 1973.

[3] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, March 1989.

[4] J. Crampton, G. Loizou, and G. O'Shea. Evaluating access control. Technical Report BBKCS-9905, Birkbeck College, University of London, 1999.

[5] H. Custer. *Inside Windows NT*. Microsoft Press, Redmond, Washington, 1993.

[6] S.K. Debray, D.S. Warren, S. Dietrich, F. Pereira, and D. Spinellis. *The SD-Prolog System, Version 3.1*. Department of Computer Science, University of Arizona.

[7] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[8] A. Heydon, M.W. Maimone, J.D. Tygar, J.M. Wing, and A.M. Zaremski. Miró: Visual specification of security. Technical Report CMU-CS-89-1989, Carnegie Mellon University, Pittsburgh, PA, 1989.

[9] G.E. Hughes and M.J. Cresswell. *An Introduction to Modal Logic*. Methuen and Co Ltd, London, 1968.

[10] A. Hunter and P. McBrien. Default databases: Extending the approach of deductive databases using default logic. *Data & Knowledge Engineering*, 26:135–160, 1998.

[11] B.W. Lampson. Protection. *ACM Operating Systems Review*, 8:437–443, 1974.

[12] M. Levene and G. Loizou. A modal logic formalism for distributed and parallel knowledge bases. *Parallel Algorithms and Applications*, 1:11–27, 1993.

[13] M. Levene and G. Loizou. *A Guided Tour of Relational Databases and Beyond*. Springer-Verlag, London, 1999.

[14] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, London, 1984.

[15] J. McLean. Reasoning about security models. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 123–131, 1987.

[16] J. McLean. The algebra of security. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 2–7, 1988.

[17] Microsoft. *Windows NT Resource Guide*. Microsoft Press, Redmond, Washington, 1995.

[18] C.G. Morgan. Methods for automated theorem proving in nonclassical logics. *IEEE Transactions on Computers*, C-25(8):852–862, August 1976.

[19] S. Moritsugu and DTR Business Systems. *Using UNIX*. QUE, Indianapolis, Indiana, second edition, 1998.

[20] G. O'Shea. On the specification, validation and verifiction of security in access control systems. *The Computer Journal*, 37:437–448, 1994.

[21] G. O'Shea. Redundant access rights. *Computers & Security*, 14:323–348, 1995.

[22] G. O'Shea. *Access Control in Operating Systems.* PhD thesis, Birkbeck College, University of London, July 1997.

[23] M.D. Schroeder, D.D. Clark, and J.H. Saltzer. The MULTICS kernel design project. *ACM Operating Systems Review*, 11:43–56, 1977.

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

[25] D. Solomon. *Inside Window NT.* Microsoft Press, Redmond, Washington, second edition, 1998.

[26] A. Thayse, editor. *From Modal Logic to Deductive Databases: Introducing a Logic Based Approach to Artificial Intelligence.* John Wiley & Sons, Chichester, England, 1989.

[27] US Department of Defense. Trusted computer system evaluation criteria. Technical Report CSC-STD-002-85, Department of Defense Computer Security Centre, Fort George G. Meade, MD, 1985.

## Appendix A: Basic Theoretic Results

**Lemma A.1** *For all $n \in \mathbb{N}$,*

*1 $I_n$ is finite for finite $W$,*

*2 $I_n$ is monotonically increasing.*

**Proof:**

1 $\Xi^K$ is finite because $\Xi$ contains a finite number of predicates and the domain of discourse, $D$, is finite. Furthermore, there are no functional constants in our logic, and the Domain Closure Assumption applies. Hence $I_n \subseteq \Xi^K$ is finite.

2 By construction, $I_{n-1} \subseteq I_n$.

■

**Corollary A.1** *There exists an integer $n$ such that $I_n = I_{n+1}$. That is, $I_n$ is a fixpoint of the consequence operator $IC$.*

**Proof:** If follows immediately from Lemma A.1. ■

**Definition A.1** *Let $N$ be the least integer such that $I_N$ is a fixpoint of the consequence operator $IC$. We call $I_N$ the least fixpoint of $IC$.*

**Theorem A.1** *$I_N$ is unique.*

**Proof:** (By induction) $I_0 = \emptyset$ by definition. Suppose that $I_n$ is unique for all $n \leqslant k$. By definition, $I_{k+1} = IC(I_k)$. Since $IC$ is a function, by induction, $I_{k+1}$ is unique. (Suppose that $IC(I_k) = I$ and $IC(I_k) = J$ with $I \neq J$. Without loss of generality we can choose $L \in I \setminus J$. Then there exists a rule $L \leftarrow L \wedge L_1 \wedge \cdots \wedge L_k$ such that $\{L_1, \ldots, L_k\} \subseteq I_k$ but, since $L \notin J$, $\{L_1, \ldots, L_k\} \nsubseteq I_k$.) ■

**Theorem A.2** *$I_N$ can be determined in polynomial time.*

**Proof:** From Lemma A.1, $I_N$ is finite. For a predicate $P$ in $\Xi$, there are at most $xy$ possible predicate forms in $I_N$, where $x$ is the number of terms in $D$ and $y$ is the arity of $P$. Since the number of predicates in $D$, the number of inference rules in $\Xi$ and the number of worlds are all finite, it follows that $I_N$ is bounded by a polynomial in the size of $D$, $\Xi$ and $W$. Furthermore, since $I_{n-1} \subseteq I_n$ for all $n \leqslant N$, it follows that $I_N$ can be determined in polynomial time. The result now follows. ■