Evaluating and Improving Access Control

Jason Crampton*  George Loizou  Greg O'Shea

Birkbeck College,  Microsoft Research Ltd,
University of London,  St George House,
Malet Street,  1 Guildhall Street,
London, WC1E 7HX, England  Cambridge, CB2 3NH, England
e-mail: {ccram01, george} @dcs.bbk.ac.uk  e-mail: gregos@microsoft.com

27 November 1999

Abstract

Our recent work provides a theoretical basis for the development of tools for reasoning about the operational implications of a particular configuration of the access control mechanism of an operating system. Herein we introduce a set-theoretic model of an access control policy and the concept of consistency of the state of an access control mechanism with a given access control policy. Our earlier work coupled with this definition of consistency enables us to assess and hence improve the implementation of an access control policy by using an access control mechanism. We demonstrate the value of our approach by specifying a simple access control policy and implementing the policy on two different commercial operating systems.

1 Introduction

The access control mechanism (ACM) provided by a multi-user operating system is widely used as a means of implementing the security requirements of the administrators of such a system. Typically it is used to prevent or restrict the access of individuals and groups of users to certain areas of a file system.

The complexity of modern ACMs, coupled with the large numbers of users and files involved, make it extremely difficult to assess whether the implementation of an ACM, or changes to an existing one, will result in the intended security configuration of the operating system and file structures.

In our recent paper [1] we investigated the use of modal logic to implement a deductive database [4] in Prolog, and to use this as a means of both representing the implementation of an ACM, and reasoning about the consequences thereof. Our results provide a basis for developing automated tools that will enhance the understanding of, and provide a means of assessing the implications of, a particular implementation of an ACM.

We believe that by offering such utility we provide an opportunity to improve operating system security. However, we must consider what it is that a particular ACM is intended to achieve, and then ascertain whether our improved ability to reason about the effects of changes made to the ACM enables us to improve its implementation. Specifically, we need to specify an access control policy (ACP) which can be used as the benchmark for the security afforded by the ACM.

We also note that there are an increasing number of heterogeneous networks that use several different operating systems. We believe the ability to specify a network-wide ACP and to assess

* Supported by EPSRC Award 98317878
implementations of that policy using ACMs on different operating systems will be invaluable to systems administrators.

The motivation, therefore, for this paper is: "does our ability to reason about the effects of changes made to the ACM enable us to more accurately and readily realise the intentions of an ACP?". Herein we introduce the notion of consistency of an ACM with the ACP it is intended to implement. This enables us to compare intended security (the requirements of the ACP) and actual security (as afforded by the ACM).

In general, there are many ways of configuring an ACM in order to implement an ACP through different choices of directory structure and user groups, and through features of the ACM employed. If we can show that an implementation is not consistent, and, furthermore, why it is not consistent with the ACP, then we provide the opportunity to improve that implementation.

In addition, we introduce a set-theoretic model of an access control policy and construct several simple examples of ACMs to illustrate the generality of our approach.

In order to demonstrate our claim that we can now identify shortcomings of an implementation of an ACP, we extend one of these examples and implement the ACP through the ACMs of the UNIX and Windows NT operating systems.

The remainder of the paper is arranged as follows. Section 2 describes our model of the ACM of an operating system, defines an interpretation of an ACP and hence introduces the idea of consistency. It also provides a motivation and theoretical basis for the material in Section 3 which presents our models of the UNIX and Windows NT ACMs and how we construct an interpretation from a given ACP. Section 4 discusses our experimental work, in which we describe a simple ACP, construct ACMs to implement that ACP, generate an interpretation in terms of UNIX and Windows NT and assess the consistency of the ACMs with respect to the ACP. Finally, in Section 5, we present our conclusions.

2 Preliminaries

2.1 Modelling the Access Control Mechanism of an Operating System

Our model is based on Lampson's protection matrix [3]. We assume that an operating system has an embedded reference monitor, or access control mechanism which controls access by subjects to objects [4]. Typically an ACM is implemented by assigning permissions to a file, as in UNIX, or through the use of access control lists, as in Windows NT.

Definition 2.1 Let $O$ be a set of passive entities, called objects, and $S$ be a set of active entities, called subjects, which act on objects. Let $R$ be a set of rights describing how objects can be accessed by subjects.

The set $F = O \times S \times R$ models the set of all possible interactions in an operating system. A member $(o, s, r) \in F$ is referred to as an access right triple or, more usually, as an access right or triple.

Definition 2.2 The state, $M \subseteq F$, of an ACM is the set of access rights which are granted by the ACM.

For example, if $M = \emptyset$ we have the most restrictive possible ACM, and if $M = F$ the least restrictive.

Definition 2.3 A computation is an event that occurs at the Trusted Computing Base [11] of the operating system, and which initiates the invocation of one or more access rights. We define the set $Q \subseteq F$ to be the set of computations, or requests, performed by an operating system. Each computation, $q \in Q$, gives rise to a set of triples, the computational closure of $q$, $T_q \subseteq Q$, which is the set of access rights required for the computation to complete. A computation, $q$, completes successfully if, and only if, $T_q \subseteq M$. For a set of requests, $A \subseteq Q$, we define $T_A = \bigcup_{a \in A} T_a$. 

2
A full account of our model of an ACM and the methods used to construct the set $T_q$ is given in [1].

In UNIX, a request, $q$, to execute a file, /test.sh, by a user, u, can be regarded as the triple (/test.sh, u, x). This gives rise to

$$T_q = \{(/test.sh, u, x), (/test.sh, u, r), (/, u, x)\},$$

where / is the root directory, x denotes “execute” access and r denotes “read” access. Given the following output from the `ls` program we can see that the request $q$ completes successfully if u is a member of the group root.

```
/    root    root    drwxr-xr-x
/test.sh  root    root    -rwxr-xr-x
```

If, in addition, the file /test.sh requires the user to write to the root directory, /, then

$$T_q = \{(/test.sh, u, x), (/test.sh, u, r), (/, u, x), (/, u, w)\},$$

in which case the request completes successfully if u is the root user.

**Definition 2.4** The effective state, $M_E$, of an ACM is defined to be those requests which complete successfully. In other words,

$$M_E = \{q \in Q : T_q \subseteq M\},$$

where $M$ is the state of the ACM.

### 2.2 Comparing an ACM to an ACP

We now consider the relationship between an ACM and an ACP. An ACP specifies those states of the ACM that preserve certain desirable properties of information confidentiality, integrity and availability [7]. We assume that the specification of an ACP includes a description of at least one of the following:

- the circumstances which are not permitted to arise (preserving confidentiality and integrity), and
- the circumstances which must always prevail (guaranteeing availability).

**Definition 2.5** An interpretation (with respect to $F$) of an ACP is a representation of the ACP in terms of requests, $q \in Q$.

An interpretation comprises sets of triples that are to be prohibited or sets of triples which are required, or both. It should be noted that if $P$ is a prohibited set of triples, then all $P'$ such that $P \subseteq P'$ are also prohibited sets of triples. Therefore, for each set of prohibited triples, there is a minimal set which can, and will in the sequel, be used to characterise that part of the ACP. In other words, if the ACP prohibits a certain combination of sensitive access rights, $P$, it does not prohibit any subset of that combination of rights (see Example 2.1. (Similarly, required sets will be assumed to be maximal.)

**Definition 2.6** $P_{ACP}$ denotes the set of prohibited sets of triples and $R_{ACP}$ denotes the set of required sets of triples.

**Definition 2.7** A state, $M$, of an ACM is consistent with (or satisfies) an ACP if, and only if,

$$\text{for all } P \in P_{ACP}, P \nsubseteq M, \text{ and there exists } R \in R_{ACP}, \text{ such that } T_R \subseteq M. \quad (1)$$

A state, $M$, of an ACM is weakly consistent with an ACP if, only if,

$$\text{for all } P \in P_{ACP}, T_P \nsubseteq M, \text{ and there exists } R \in R_{ACP}, \text{ such that } T_R \subseteq M. \quad (2)$$
In other words, no consistent state contains a prohibited set, and every consistent state contains the computational closure of at least one required set. (No weakly consistent state contains the computational closure of any prohibited set. In operational terms, weak consistency is sufficient to realise the requirements of an ACP as it prevents every triple in a prohibited set from completing successfully. It is debatable whether one would use an ACM that was known to be weakly consistent because irreparable damage may already have been done during the course of a computation that does not complete successfully.)

We believe that in practice it will be possible to identify a single required set of triples, \( R \), as the intended interpretation of (that part) of the ACP, and that each element of \( P_{ACP} \) will usually be a set consisting of a single triple (that is, \( P_{ACP} \) is essentially a set of prohibited triples, \( \{\{q_1\}, \{q_2\}, \ldots, \{q_s\}\} \)). In such circumstances (1) can be re-stated as

\[
\text{for all } \{q_i\} \in P_{ACP}, q_i \notin M, \text{ and } T_R \subseteq M,
\]

and (2) as

\[
\text{for all } \{q_i\} \in P_{ACP}, T_R \notin M, \text{ and } T_R \subseteq M.
\]

**Definition 2.8** If \( P \) is a prohibited set of triples and \( T_p \subseteq M \), we say \( P \) is a vulnerability in \( M \). If \( M \) does not contain the computational closure of any required set, we say \( M \) is deficient.

**Definition 2.9** If \( R \) is a required set of triples and \( T_R \subseteq M \), the members of \( M \setminus T_R \) are redundant access rights.

Intuitively, redundant access rights are those which are granted by the ACM but which are not required to implement the ACP.

**Example 2.1** Consider the following simple scenario. There are two executable files, \( f_1 \) and \( f_2 \), and two users, \( u_1 \) and \( u_2 \). The only access right supported by the operating system is “execute”, which we denote by \( x \). The statement of the ACP is

[1] no user is permitted to execute both files, and \[2] both files must be executed.

In other words, this is a trivial example of an ACP which requires the division of responsibility. Then

\[
F = \{(f_1, u_1, x), (f_2, u_1, x), (f_1, u_2, x), (f_2, u_2, x)\},
\]

\[
P_{ACP} = \{(f_1, u_1, x), (f_2, u_1, x), (f_1, u_2, x), (f_2, u_2, x)\} \text{ from [1]},
\]

\[
R_{ACP} = \{(f_1, u_1, x), (f_2, u_2, x), (f_1, u_2, x), (f_2, u_1, x)\} \text{ from [2]}.
\]

Although there are two members in the set \( R_{ACP} \), one would expect to be able to identify which represents the intended security of the operating system as each of the two users will be identified with a particular role (and hence which file they should execute) by other requirements of the ACP.

**Example 2.2** Consider the following scenario. There are three objects, a paper file, \( f_p \), a script file, \( f_s \), and a result file, \( f_r \). There are two users, a lecturer, \( u_l \), and a student, \( u_s \). The ACP states that

[1] the lecturer has full permission to access all files,

[2] while the student can read . . .

[3] . . . but not write to the paper file,


[5] . . . and is denied all access rights to the result file.
Let us assume that we will interpret this ACP in the context of a UNIX operating system, which supports "read", r, "write", w, and "execute", x, access.

\[ P_{ACP} = \{(f_p, u_s, w)\}, \text{ from [3]} \]
\[ \{(f_s, u_s, w), (f_r, u_s, r)\} \text{ from [5]} \]

\[ R_{ACP} = \{(f_p, u_1, w), (f_p, u_1, r), \text{ from [1]} \}
\{ (f_s, u_1, w), (f_s, u_1, r), \text{ from [4]} \}
\{ (f_r, u_1, w), (f_r, u_1, r), \text{ from [1]} \}
\{ (f_p, u_s, r), \text{ from [2]} \}
\{ (f_s, u_s, w), (f_s, u_s, r)\} \text{ from [4]} \]

It should be noted that condition [1] seems to imply that the triples \((f_p, u, x), (f_s, u, x)\) and \((f_r, u, x)\) should be in \(R_{ACP}\); and condition [5] that the triple \((f_r, u_s, x)\) should be in \(P_{ACP}\). However, these triples have no operational meaning (we cannot "execute" a "plain" file), and hence these triples do not belong to \(Q\), the set of computations performed by this operating system.

We can now implement this ACP by assigning file permissions to \(f_p\), \(f_s\), and \(f_r\) and creating user groups.

3 Practical Considerations

We now detail the methods we adopt in conducting our experiments. Our broad strategy is as follows, and is described in more detail in the following sections.

- Specify the ACP.
- Construct the ACM (intended to implement the ACP) of an operating system by specifying the file system and user groups.
- Infer the state of the ACM.
- Construct the interpretation of the ACP with respect to the relevant operating system.
- Infer the sets \(R_{ACP}\) and \(P_{ACP}\).
- Derive the computational closure of the set of required triples and of each of the prohibited sets using our deductive database. (A detailed account of the deductive database used to reason about the computations that are permitted by a given state of the ACM can be found in [1] and [7].)
- Assess the consistency or otherwise of the state of the ACM with respect to the ACP.

Our earlier work [1] used the Prolog programming language, and we continue to employ it in this paper. We do not present the source code in full, preferring to give illustrative examples from it.

We denote a predicate \(p\) of arity \(n\) by \(p/n\), and note the following syntactic conventions of Prolog:

- The empty list is denoted by \([]\).
- An anonymous variable is denoted by the underscore character, _
- Variable names begin with an upper-case letter and constant names begin with a lower-case letter or are enclosed in single quotes.
3.1 Specifying an Access Control Policy

Various specification schemes for ACPs exist in the literature which give rise to different types of policy. A comparison of these, in terms of how the choice of specification can affect the implementation and validation of an ACM, can be found in [6].

We will, however, specify our ACP in English as our experiments concern a simple scenario and our main focus is to compare the interpretation of an ACP with an ACM. (It is clear that subsequent work will need to consider how a more formal specification of an ACP can be interpreted with respect to a given operating system.)

We assume for the purposes of this paper that we can write down statements which have an unambiguous meaning and which capture precisely the requirements of our ACP.

3.2 Describing the Windows NT Access Control Mechanism

The ACM in Windows NT [8] is implemented by assigning security descriptors to objects (including files and directories). A security descriptor includes the security ID (SID) of the owner of the object and a discretionary access control list (DACL).

A DACL is comprised of access control entries (ACEs). An ACE either allows or denies an individual or a group the access rights listed in its access mask. (An object that has no DACL allows full access to all users; a DACL that has no ACEs denies all access to all users.)

We model users with the predicate user_def/1, and groups with the predicate group_def/2. Typical examples are shown below.

```
user_def(jc).
group_def(students, [jc, gg]).
```

We model ACEs with the predicate ace/3. ACEs will be represented by facts of the following form,

```
ace(Mode, User, Rights)
```

where Mode ∈ \{a, d\} respectively denoting “allowed” and “denied”; either user_def(User) or group_def(User, _ ) is true, and Rights is a non-empty list of elements from the set

\{r, w, c, x, d, a, p, o, all\}

which respectively denote “read”, “write”, “change”, “execute”, “delete”, “add”, “change properties”, “take ownership” and “all rights” access.

We model files with no DACL using the predicate file_def/2, and those with a DACL using the predicate file_def/3. Files will be represented by facts of the following form,

```
file_def(File_Name, Owner, dacl(ACEs))
```

where ACEs is a list of access control entries. (If ACEs is the empty list, then access is denied to all users.)

Windows NT determines the access rights to a file as follows. If the user is either the owner or has take-ownership privileges the user is granted “change properties” access. The user is then granted all access rights that appear in an access-allowed ACE (relevant to that user in the DACL) that do not appear in an access-denied ACE (relevant to that user).

3.3 Describing the UNIX Access Control Mechanism

The ACM in UNIX [5] is implemented by allocating access rights to a file to three sets of user accounts, namely the owner of the file, the owner group of the file and other users. The union of the members of these three sets comprises the entire set of users. The access rights of the owner override those of the owner group which in turn override those of other users. (We note that certain commercial versions of UNIX now include more sophisticated methods, including access
control lists, in their ACMs [2, 9, 10]. We also note that there are additional options on the execute bit of file permissions, which provide more flexibility in the specification of access to directories and to executable files.)

We model UNIX files using the predicate file_def/4. A typical example is shown below.

\[
\text{file_def('/example', root, lecturers, \[d,r,w,x,r,w,x,r,-,x\])}.
\]

The intended meaning of the above clause is that “the file /example is owned by user root and by the group lecturers”. (The file permissions are represented as a list, rather than a string, to facilitate processing.)

We model UNIX accounts using the predicate user_def/2. A typical example is shown below.

\[
\text{user_def(jc, \[\text{students, java}\])}.
\]

The intended meaning of the above clause is that “the UNIX account jc belongs to the groups students and java”. The primary group of the user, that is, the default owner group of a file created by the user, is the first element in the list.

We also make use of the predicate file_def/3 of the form

\[
\text{file_def(F, Owner, acl([\{(a, Owner, R1), (a, Group, R2), (a, Other, R3)\}]}}).
\]

This is derived from file_def/4 in the obvious way. This predicate is the same as that used to represent files in Windows NT, enabling us to use a common method for inferring the state of the ACM.

### 3.4 Inferring the State of the ACM

We infer the rights granted to users of a file by using the predicates expand_dacl/2, expand_dacl/4 and expand_ace/2. These predicates recursively expand the ACEs in the DACL first by the group if appropriate and then by the access mask. In other words, the original ACE is replaced by a list of triples of the form ace(Node, User, Right), where user_def(User) is true.

A side-effect of recursively expanding the DACL is to partition it into those triples which are allowed and those which are denied. When the entire DACL has been expanded, the complement of the allowed triples with respect to the denied triples is returned as the set of triples which are allowed by the DACL. This is expressed in the base case of the recursion for expand_dacl/4 as follows.

\[
\text{expand_dacl([], Denied, Allowed, Access) :-}
\]

\[
\quad \text{complement (Allowed, Denied, Access).}
\]

This provides a framework for enumerating the access rights that are granted by the Windows NT ACM, and hence for deducing the state, M, of such an ACM.

### 3.5 Interpreting an Access Control Policy

The interpretation of the ACP is by definition composed of access right triples. Two predicates, required/3 and prohibited/3, generate such triples, and are defined in terms of the predicates object/2 and subject/2. A typical definition (for these predicates with respect to a UNIX ACM) is shown below.

\[
\text{prohibited(File, User, w) :-}
\]

\[
\quad \text{object(File, paper),}
\]

\[
\quad \text{subject(User, student).}
\]

\[
\text{object(File, paper) :-}
\]

\[
\quad \text{file_def(File, _, _, [ - | _ ]),}
\]

\[
\quad \text{parent_dir('/papers', File).}
\]
\text{subject}(\text{User}, \text{student}) :-
  \text{user_def}(\text{User}, \text{Groups}),
  \text{member}(\text{students}, \text{Groups}).

The intended semantics of the above clauses is that “no student should be able to amend an
examination paper” assuming a suitable file structure has been defined.

Naturally the definition of the predicates \text{object}/2 and \text{subject}/2 will depend upon the ACP
and the ACM through which the ACP will be implemented.

Clearly, in the example above, we do not explicitly need the predicates \text{object}/2 and \text{subject}/2.
We could simply write the rule as shown below, but their use makes the code more readable and
easier to maintain.

\text{prohibited}(\text{File}, \text{User}, w) :-
  \text{file_def}(\text{File}, _, _, [\text{-} | \text{-}]),
  \text{parent_dir}('papers', \text{File}),
  \text{user_def}(\text{User}, \text{Groups}),
  \text{member}(\text{students}, \text{Groups}).

This provides us with a means of inferring the members of the sets $P_{ACP}$ and $R_{ACP}$.

3.6 Computing the Set $T_R$

The deductive machinery introduced and discussed in detail in [7] can be invoked to deduce the set
$T_R$ from $R_{ACP}$. Briefly, this machinery consists of facts and rules defining the \text{needs}/2 predicate.
Typical examples in the context of a UNIX ACM are shown below.

\text{needs}((\text{File}, \text{User}, x), (\text{File}, \text{User}, r)).
\text{needs}((\text{File}, \text{User}, _), (\text{Dir}, \text{User}, x)) :-
  \text{parent_dir}(\text{Dir}, \text{File}).

The semantics of the first clause are that “for a user to execute a UNIX file, it is necessary that
the user has read access to that file”. The semantics of the second are that “for a user to have
any kind of access to a file, it is necessary that the user has execute access to the parent directory
of that file”.

These clauses are derived from an abstract understanding of the dependencies of access rights
in a given ACM, and in a real-world application, from empirical evidence provided by auditing
the operating system.

3.7 Determining the Consistency of the ACM with the ACP

We can now determine whether or not $M$ is consistent with the ACP. Having computed the sets
$P_{ACP}$, $R_{ACP}$, $T_R$ and $M$ it remains to check the intersection properties of Definition 2.7 to see
whether the state $M$ of the ACM is consistent with the given ACP.

4 Experimentation

We use a single example of an ACP and implement it on our models of the ACMs of UNIX
and Windows NT. Our example is restricted to considering the allocation of file permissions to
directories and files. Its purpose is to demonstrate that our methods enable us to highlight where
the state of an ACM is not consistent with the ACP it aims to implement, and, therefore, that we
have a valuable tool for prototyping ACMs prior to their actual implementation.

We note that it is not our intention to demonstrate that we can configure an ACM correctly
(or otherwise!).
4.1 Description

Our ACP concerns the writing of exam papers, and submission and marking of scripts in the Computer Science Department of a university. There are two groups of users of the computer system, lecturers and students. We assume for the purposes of this example that the sets of students and lecturers are disjoint.

We wish to incorporate the following security features into the system. We annotate ACP statements regarding availability, confidentiality and integrity with A, C and I, respectively.

- A lecturer should be able to set, amend and mark exam papers in subjects in which (s)he runs a lecture course. (A)
- Any lecturer should be able to read any exam paper. (A)
- A student sits exams by submitting a script. To do this (s)he must (be able to) read the exam paper. (A)
- A student can only sit exams for which (s)he is registered. (I)
- No student can write to an exam paper. (I)
- No student can have any access to a script other than one (s)he owns. (C, I)
- Lecturers who have run a lecture course mark the scripts. (A)
- Marks are written, by the marker, to result files. (A)
- Any lecturer should be able to read any result file. (A)
- No student has any access to the result files. (C, I)

4.2 Implementing the ACP on an Operating System

We assume that there are two courses, two lecturers, and two students. In the cases of the lecturers and students, one of each is involved with both courses and one with only one. This is sufficient to demonstrate that the requirements of the ACP are satisfied (or otherwise), and that the implementation could be extended to any number of courses, lecturers and students.

We realise the requirements of this ACP by having four groups of users representing lecturers, students, people involved with Java and people involved with Ada (where “people involved with” means lecturers who run the course and students who study the course). There are directories for each of the courses, each of which has three sub-directories containing papers, scripts and results.

The basic mechanism we choose for achieving the requirements of the ACP is to make a course directory available only to members of the group representing that course, and to make the sub-directories available as appropriate to the members of either the lecturer or student groups.

The file definitions and user group definitions for the configuration of the UNIX ACM are shown below. We note that there are likely to be many other ways of implementing this ACP (through different choices of file structure and definitions of user groups). Although the basic directory and group structure has not changed from our original implementation, several refinements have been made to the file permissions as a result of evaluating the consistency of earlier implementations. This, in fact, is the basis of our thesis. Namely, the ability to assess the effectiveness of actual security compared with intended security enables systems administrators to make better decisions.

We also note that we amended the original ACP so that the sets of lecturers and students were disjoint. This was because it was found to be impossible to maintain confidentiality of papers and marks without significantly changing the user group structure. By considering another configuration of the ACM where each course had a lecturer and a student group, we were able to realise the original ACP, but at the operational cost of doubling the number of user groups.

We note the following points.
file_def('', root, root, [d,r,w,x,r,-x,-x,-x]).
file_def('/java', root, java, [d,r,w,x,r,-x,-x,-x]).
file_def('/java/papers', root, lecturers, [d,r,w,x,r,w,x,r,-x]).
file_def('/java/papers/gl_paper', gl, lecturers, [-r,w,-r,-r,-r,-r,-r,-r]).
file_def('/java/papers/rg_paper', root, lecturers, [d,r,w,x,r,-x,-x,-x]).
file_def('/java/scripts/jc_script', jc, students, [-r,w,-r,-r,-r,-r,-r,-r]).
file_def('/java/scripts/gl_script', gg, students, [-r,w,-r,-r,-r,-r,-r,-r]).
file_def('/java/results', root, lecturers, [d,r,w,x,r,x,-x,-x]).
file_def('/java/results/gl_marks', gl, lecturers, [-r,w,-r,-r,-r,-r,-r,-r]).
file_def('/java/results/rg_marks', root, lecturers, [d,r,w,x,r,-x,-x,-x]).
file_def('/java/results/gl_marks', gl, lecturers, [-r,w,-r,-r,-r,-r,-r,-r]).
file_def('/java/results/rg_marks', root, lecturers, [d,r,w,x,r,-x,-x,-x]).
user_def(root, [root]).
user_def(gl, [lecturers, java, ada]).
user_def(jc, [students, java, ada]).
user_def(rg, [students, java]).

Figure 1: Prolog facts for UNIX implementation

- The root directory is represented as the empty string, rather than the usual /, because of the semantics of our predicate parent_dir. However, it can be seen that the name of a subdirectory in UNIX is formed by concatenating the name of the parent directory with a forward slash and with the name of the subdirectory. Hence the directory /java, for example, is formed by concatenating the strings “”, “/” and “java”. That is, our representation is consistent.

- Students are denied read access to a script directory, so that they cannot list the contents of such directories and find other students’ submissions. This condition was introduced as a result of our experimentation which indicated that students would be able to read each other’s scripts (It is worth noting that even these simple implementations of our ACP have been refined by our ability to compare intended and actual security.)

- We assume that no user amends the file permissions of a file (s)he owns, in order to protect it from other users. We also assume that a new file’s permissions are given by the umask setting 033 [5]. (In the context of this example, there is a strong case for students being prohibited from amending file permissions. In other words, they should not be able to execute the chmod program. In Windows NT we could implement this requirement by including, for example, ace(d, jc, [p]) in the DACL for the file /java/scripts/jc_script.)

The definitions of the file system and groups for Windows NT (which appear less familiar and are less easy to assimilate) are shown in Appendix A.

4.3 Results of the UNIX Implementation

Table 1 shows the output from the Prolog interpreter for the prohibited sets of triples in the interpretation of the UNIX implementation, and whether these sets fail to satisfy the criteria for
consistency and weak consistency, respectively.

<table>
<thead>
<tr>
<th>Prohibited Set, $P$</th>
<th>$P \subseteq M$</th>
<th>$T_P \subseteq M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>{(/java/scripts/gg_script, jc, r)}</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>{(/java/scripts/jc_script, gg, r)}</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>{(/ada/scripts/jc_script, gg, r)}</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>{(/ada/results/rm_marks, gg, r)}</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>{(/ada/results/rm_marks, jc, r)}</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>{(/ada/results/gl_marks, gg, r)}</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>{(/ada/results/gl_marks, jc, r)}</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>{(/java/results/gl_marks, gg, r)}</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>{(/java/results/gl_marks, jc, r)}</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>{(/java/results/gl_marks, gg, w)}</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>{(/java/results/gl_marks, jc, w)}</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>{(/java/results/gl_marks, gg, w)}</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>{(/java/results/gl_marks, jc, w)}</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>{(/ada/papers/rm_paper, gg, w)}</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>{(/ada/papers/rm_paper, jc, w)}</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>{(/ada/papers/gl_paper, gg, w)}</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>{(/ada/papers/gl_paper, jc, w)}</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>{(/java/papers/gl_paper, gg, w)}</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>{(/java/papers/gl_paper, jc, w)}</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 1: The results of the UNIX experiment

We see that the implementation is not consistent with the ACP even in the weaker sense. It is clear that the problem lies in the fact that if a student creates a script, then by default the group associated with that file is the group students. Any student who studies the course will be able to read that script if (s)he knows the name of it. (S)he is, however, prevented from listing the directory as noted earlier. (A student who is not on the same course will not be able to read the script because of the group ownership of course directories.)

Thus our example highlights a vulnerability in our implementation. One possible solution would be to change the umask setting (which controls the allocation of permissions to a file at creation time [5]) for students in such a way that files they create permit no access to members of their group.

We will not list the required triples as there are in excess of 250 of them, but note that the UNIX implementation is also inconsistent because, for example, a lecturer on the Ada course cannot write to papers owned by the other lecturer. (This problem can also be solved by using the umask command.)

4.4 Results of the NT Implementation

The NT implementation was shown to be consistent. DACLs can easily be constructed to prevent a student from reading another student’s script, even if they follow the same course. For example
an appropriate DACL for the file '/java/scripts/jc_script' would be

\[\text{dacl}([(a, jc, [all]), (a, lecturers, [r])])\]

The greater degree of flexibility in defining DACLS in NT is always more likely to enable us to engineer a consistent state of the Windows NT ACM. Not only do DACLS offer the opportunity to deny access rights to (groups of) users (an option we did not exercise in this experiment), they also offer complete freedom in defining the users and groups of users that appear in the ACEs. UNIX, however, insists that permissions be allocated to each of three groups whose union comprises the whole user population.

5 Conclusions

We have shown that it is possible to evaluate the security afforded by a prototypical implementation of an ACM with respect to that intended by the ACP. We have also found that the techniques outlined in this paper are constructive in the sense that they highlight vulnerabilities and deficiencies in the implementation of the ACP. If we were to couple this work with an analysis of the invocation of access rights in an existing implementation of an ACM, as we have done in earlier experiments, we would have a methodology for analysing and improving the implementation of ACPs in operating systems.

We recall our original motivation for this paper: “does our ability to reason about the effects of changes made to the ACM enable us to more accurately and readily realise the intentions of an ACP?” We would argue, therefore, that we can answer this question in the affirmative, and that our definition of consistency provides a meaningful and constructive way of evaluating the security provided by an ACM given a formal specification of security in the form of an ACP.

The experiments raise (or leave unanswered) the following questions.

- How well will these techniques scale?
- Is it possible to construct an interpretation that faithfully represents the ACP?

To answer either of these questions we will need to provide more formal methods for specifying an ACP. We believe previous attempts to tackle the specification of ACPs have either been in the context of new operating systems, which will have the ACP “hard-wired” into the operating system, or in the context of a particular existing operating system [7]. We would like to specify ACPs which may be implemented through the ACM of any operating system of interest, and believe that new methods may need to be found. Given that we employ logic programming, we are obviously drawn to the possibility of using first order logic to provide such specification methods.

We also need to find a more systematic way of generating an interpretation of a given ACP on a given platform. We did not incorporate into our experiments, for example, the fact that in UNIX file permissions are controlled by the ability of the user to execute the chmod program (while in Windows NT we could exercise an explicit option in the creation of an ACE). One possibility that presents itself is the use of an “ideal” ACM that can model the ACM of any operating system of interest; and within whose context any ACP can be interpreted and hence mapped onto the ACM of a real-world operating system. This, of course, is a matter for further research.

In connection with the above we will be investigating the following questions and the computational complexity of computing an answer.

- Is it possible, in general, to construct an interpretation for a given ACP with respect to a given operating system?
- Does there exist a state of an ACM for a given operating system which is consistent with a given ACP?
- Is a given ACP or its interpretation self-consistent?
We also intend to investigate further the properties of the effective state of an ACM. We believe the set of triples of an effective state may provide a more efficient way of building ACMs that are consistent with a given ACP.

References


Appendix A: NT Implementation

To simplify the coding of the experiment, we use the same file-naming convention as that for UNIX files rather than, for example, 'c:/java'.

file_def('', admin, dacl([[a, students, [r,x]], (a, lecturers, [r,x]))]).

file_def('/java', admin, dacl([[a, java, [r,x]]])).
file_def('/java/papers', admin, dacl([[a, lecturers, [r,x,a]], (a, students, [r,x]]))).
file_def('/java/papers/gl_paper', gl, dacl([[a, lecturers, [r]], (a, gl, [all]), (a, students, [r])])).
file_def('/java/scripts', admin, dacl([[a, students, [a]], (a, lecturers, [r,x])])).
file_def('/java/scripts/gl_script', jc, dacl([[a, jc, [all]], (a, lecturers, [r])])).
file_def('/java/results', admin, dacl([[a, lecturers, [r,x]]])).
file_def('/java/results/gl_marks', gl, dacl([[a, gl, [all]], (a, lecturers, [r])])).

file_def('/ada', admin, dacl([[a, ada, [r,x]]])).
file_def('/ada/papers', admin, dacl([[a, lecturers, [r,x,a]], (a, students, [r,x])])).
file_def('/ada/papers/gl_paper', gl, dacl([[a, lecturers, [r]], (a, gl, [all]), (a, students, [r])])).
file_def('/ada/papers/rm_paper', rm, dacl([[a, lecturers, [r]], (a, rm, [all]), (a, students, [r])])).
file_def('/ada/scripts', admin, dacl([[a, students, [a]], (a, lecturers, [r,x])])).
file_def('/ada/scripts/jc_script', jc, dacl([[a, jc, [all]], (a, lecturers, [r])])).
file_def('/ada/results', admin, dacl([[a, lecturers, [r,x]]])).
file_def('/ada/results/gl_marks', gl, dacl([[a, gl, [all]], (a, lecturers, [r])])).
file_def('/ada/results/rm_marks', rm, dacl([[a, rm, [all]], (a, lecturers, [r])])).

user_def(jc).  group_def(lecturers, [gl, rm]).
user_def(gg).  group_def(students, [jc, gg]).
user_def(gl).  group_def(ada, [gl, jc, rm]).
user_def(rm).  group_def(java, [gl, jc, gg]).