Access Control for Semantic Information Systems

Sam Moffatt

Department of Maths and Computing
Faculty of Science
University of Southern Queensland

Δ

Thesis submitted for the Degree of Master of Computing at the
University of Southern Queensland

· 2010 ·

Abstract

Access control has evolved in file systems. Early access control was limited and didn't handle identities. Access control then shifted to develop concepts such as identities. The next progression was the ability to take these identities and use lists to control what those identities can do. At this point we start to see more areas implementing access control such as web information systems. Web information systems has themselves started to raise the profile of semantic information. As semantic information systems start to expand new opportunities in access control become available to be explored.

This dissertation introduces an experimental file system. The file system explores the concept of utilising metadata in a file system. The metadata is supported through the use of a database system. The introduction of the database enables the use of features such as views within the file system. Databases also provide a rich query language to utilise when finding information. The database aides the development of semantic meaning for the metadata stored. This provides greater meaning to the metadata and enables a platform for rethinking access control

Keywords: access control, semantic information systems, web information systems, XACML, file systems

Tip 1: Care about your craft.

The Pragmatic Programmer

Contents

List of Figures

List of Tables

Acknowledgments

To my supervisors Ron Addie and Richard Watson for working with me throughout this process.

Part I

Introduction

Chapter 1

Introduction

1.1 Overview

Access control today presents two significant problems. The first is the overwhelming complexity which daunts the average user. The second problem is the issue of scaling. Some systems provide sophistication at the expense of rarely used complexity. Other systems provide limited flexibility which is further hampered by user interaction issues. A method needs to be developed that presents a middle ground. The ability to define access control rules without compromising flexibility but while also being able to apply to a large amount of information easily.

This dissertation aims to cover existing systems and their behaviours looking at how they operate and issues with the systems as they currently stand. This is done through a combination of literature survey, historic review and practical experimentation of modern file systems. The dissertations also introduces an experiment in metadata file systems, MDFS, and covers its purpose, specification, design and final implementation. The conclusion of the dissertation takes time to examine current problems not otherwise identified, future problems and potential solutions in addition to areas requiring further research.

1.2 Background

1.2.1 Single User Systems

File systems are one of the most important parts of an operating system's design. File systems provide the ability to persistently store information across reboots and are now an important part of operating system security. Early operating systems only had the concept of a single user that would operate the device. This lead to file systems that did not need to record file "ownership" as it was not a feature of the operating system. The ability to define per user access controls was also absent under this model. Environments such as Microsoft Disk Operating System (MSDOS), which was a single user environment, featured a file system called "File Allocation Table (FAT)" which didn't record file ownership or user specific permissions. FAT did have some limited permissive attributes to control if a file was read-only or hidden. Files marked read-only couldn't be edited and files marked as hidden didn't appear in directory listings. Other file systems, such as Apple's Hierarchical File System (HFS) and Mac OS Extended (HFS+) have a similar option which the user can "lock" files to prevent any modifications being applied to it. Early versions of Apple's file systems (Macintosh File System (MFS) and HFS) also didn't record the owner of the file.

1.2.2 Mainframes and time sharing

Mainframe systems developed and deployed at this time introduced the concept of multiple users and time sharing. This introduced the concept of a user as a discrete entity and that multiple users could be using the system at the same time. Multiple users could also be a part of different access control levels. An implementation of this can be found in Multiplexed Information and Computer Service (MULTICS) whose security architecture was heavily influenced by the Department of Defence. MULTICS introduced many revolutionary concepts in general however in the security realm two significant features were introduced. The first is that of hardware supported access control. The "ring" system that is found on modern computers was developed with MULTICS. Rings were a method of ensuring that a lower application couldn't access the memory of a more privileged application. This could be considered as one of the first protected memory implementations. The second was the ability to "label" binaries, users and data. The classification system used for this was derived from the US Department of Defence's existing security classification model (e.g. top secret, secret, classified, unclassified, etc). A hierarchy existed where users of a lower level couldn't read data from a higher level but could create data at a higher level than their security label. The concept of read down (a more privileged user can read items at their own level and lower levels) and write up (a user can create items at their level or a higher level but not a lower level) is the enforcement of this access control model.

1.2.3 Networking

UNIX emerged as a simpler implementation of MULTICS however many of the concepts explored in MULTICS can be seen in UNIX. UNIX took parts of the existing MULTICS access model and modified it to user owners and groups for files instead of labelling files. This meant that a user could own their own files, control their own access, assign the file a group and then control its access as well as controlling access for anyone in the system. UNIX was the natural evolution from multiple users with limited access controls to multiple users with a primitive discretionary access control system (limited to three categories: self, group and everyone). Multiuser environments such as UNIX required multiuser access controls and caused the creation of the POSIX standard. POSIX.1 supports the concepts of having a user who owns a file, a group who owns a file and the ability to apply generic access control rules for everyone who is a valid user in the system. These access control models are still available in many file systems with Linux supporting POSIX standards and Windows supporting POSIX compliance modes [24] in addition to Apple's Mac OS X 10.5 being certified as POSIX compliant [27]. POSIX compliant operating systems all permit the same level of functionality with access controls. Prior to the advent of networking, multiuser time shared systems existed which provided their own access control models.

1.2.4 The Orange Book

The United States Department of Defence in 1983 issued a document called the "Trusted Computer System Evaluation Criteria", otherwise known as "The Orange Book". The Orange Book describes various levels of security from D (the system evaluated didn't meet the requirements of a higher division) to A (inclusive of the requirements of all other levels, called "Verified Protection" where the system has been formally verified to meet the requirements). With the development of the current Windows security model, Microsoft appears to have aimed to achieve the C2 compliance as defined by the criteria.

Microsoft notes that the Access Control List (ACL) model was a part of their design decisions for the Windows security system including the default Windows NT file system, NTFS [84]. The C2 security certification requires "discretionary access control" to permit the creator of the file to assign who might access the files contents and how they do this so ACLs became an important feature in NTFS. C2 defines other security behaviours as well that aren't distinctly a part of file system design such as logging requirements [85]. The UNIX world also had its own version of ACL with the draft and now abandoned POSIX.1e standard.

The POSIX.1e standard provided similar functionality to support much of the requirements of C2 level and even support for the B level which requires mandatory protection systems. Unfortunately POSIX.1e was abandoned in 1998 while still in draft form, however Mac OS X supports a modified form of POSIX.1e ACL security [16]. The New Technology File System (NTFS) security model seems to have been replicated in the development of Network File System (NFS) version 4 [93]. This model is also used with ZFS [21] as well in place of supporting POSIX.1e ACL permissions (though POSIX.1 is supported).

It appears that the model developed by Microsoft is quite robust and is subject to emulation. Microsoft has continued their work in file system research with the development of Windows Future Storage (WinFS). WinFS itself exists within the standard Windows security model. This has meant that Microsoft hasn't innovated with the file system's access control model [83]. WinFS as a project has been terminated for the time being thus it is unlikely that Microsoft will release or update anything relating to it in the near future.

1.2.5 Web Information Systems

While operating systems and file systems have a long history of the development of file systems, another area provides interesting developments in the space of access control: web information systems. As the web has shifted from something that is mostly static into something that is incredibly dynamic, the need for access controls has changed. Web information systems have permitted a myriad of different situations that need controlling of actions that a person can complete. The web progressed from being a mostly read only medium into a medium that could be used to update itself, the read-write web.

The first wave of online editable web pages appeared in the form of e-commerce solutions. Paul Graham's "ViaWeb" application permitted customers to create an online store through a completely web interface and is regarded as one of the earliest e-commerce applications on the web. The system had two basic levels of access control: an administrator account that can edit items in the store and accounts for shoppers of the system [76].

Other systems have different requirements. Consider phpbb2, a PHP based bulletin board system. phpbb2 had two major modes of functionality: a site mode and an administrator mode. Within the site, access can be granted to subforums of the main board. Access control rules can control if a user can see a forum, if they can add posts, if they can edit posts, whose posts they can edit (e.g. they may not be able to edit others posts but they can edit their own posts; or they might not be able to do this), if they can delete posts and what limitations are imposed there and if they can or can't comment on existing posts. Administrator access permitted control over the entire system including user and access management. This system demonstrates access control lists with a list of domain specific actions.

Web sites have also evolved from static websites using unchanging HTML into dynamically generated pages created from databases. These systems are known as web content management systems [54, 45, 53] and provide tools to manage editing and compositing content together to form web pages. Web content management

systems like Joomla! [48], Wordpress [55] and Drupal [46] have become popular tools for maintaining web sites.

Web content management systems have their own access control model. Initial design includes an administrator who can author content then having registered users who can submit items to the system. The final stage results having more levels of access with differing abilities in a generic sense. It includes having more control with discretionary access control to delineate fine grained control of access that are available for the particular system. A great example of this is the evolution of access control in Joomla!. Initial Joomla! releases had very little access control which slowly developed into a tiered primitive access control model in Joomla! 1.0 and now in the upcoming Joomla! 1.6 features a full discretionary ACL system.

The web has also evolved a new form of application: the social networking tool. Today large amounts of information about relationships between people are stored in web services such as Facebook. Facebook allows people to become 'friends' with people and define the sort of relationships that they have. For example you can specify someone as a friend, a co-worker or similar. This has lead to the proposal of "relation based access control" or "RelBAC" [67]. Relation based access control uses these relationships between users as a basis of defining rules for access to items.

1.3 Concepts

Within this dissertation I wish to examine some concepts about how we treat access control, permissions and identity management in various systems. However in doing so, I need to review what I mean by access control, permissions and identity management.

Many of these phrases are used in reference to a multitude of different systems each with a different meaning assigned to them.

1.3.1 Semantic Information Systems

Semantic information systems are systems in which extra metadata is associated with an object. The most prevalent example of a semantic information system is the effort to build a semantic web such as Dublin Core [70, 47]. This effort to make the web pages more descriptive in their metadata in a uniform and non-proprietary manner involves a number of different techniques of associating extra metadata with a page. Within the file system context, the push to develop metadata file systems such as WinFS [83] could be also nominated as a semantic information system. In a metadata file system, the metadata that could potentially be locked up in the contents of the file are exposed to an external interface analogous to a semantic web style system. A metadata file system could also model the capabilities that the semantic web presents within the file system context.

1.3.2 Identity Management

Identity management is the technique where a distinct entity is verifiably identified by a particular system. Entities could be intelligent agents or services that run in a system however it is more likely that the entity is a human operator. Bishop states that "identity is simply a computer's representation of an entity" [62]. Bishop defines a *principal* as a unique entity and that an *identity* specifies a principal. Identity management aims to authenticate that a particular entity is who they claim to be and is a basic requirement of any access control system. Without such a system to uniquely and verifiably authenticate and thus identify that a given entity is who they claim to be then there is no point in further access control

restrictions. An access control system which cannot verify the identity of its users can't ensure that the rules applied are appropriate. Identity management can use multiple techniques for authenticating an individual, the most popular of which is a username with a password. Other methods include certificates, smart cards, token generators and biometrics. One or more of these methods might be used to verify a persons identity however typically only a username and password is deemed sufficient. This dissertation makes the assumption that an entity is uniquely identified by some form of an identity management system.

1.3.3 Access Control

Access control is defined as a method for granting or restricting entities to complete actions on objects. Access control systems exist to provide a mechanism for ensuring access is controlled to given objects. Access control systems typically work by examining a set of rules known as "permissions" to evaluate if a entity should be permitted to complete a nominated action upon a given object. Access control systems define behaviours that should occur in particular circumstances. An example of such behaviours could be the precedence of a grant or the default grant.

1.3.4 Permissions

Permissions are the rules that make up the constraints of the access control system. Permissions in the strictest sense are a form of metadata. A permission could be described as a tuple of five fields:

entity — action — object — grant — [inheritance]

An *entity* may complete a particular *action* upon a given *object* if they are *granted* to do so which may optionally be *inherited* to child objects.

The *entity* represents something requesting to complete an action upon an object. An entity might be a user, a group, a role, a service or an autonomous agent.

An *action* is a particular task that can be completed. This could be as simple as reading, writing or deleting an object. Actions vary between system and there could be an action to complete a login.

An *object* represents the resource being protected. This could be a file, a folder or some other abstract resource that needs to be represented, such as a computer or a socket.

The *grant* describes if the rule is permissive or restrictive. A grant may either be an allow, a deny or an inherit. In some systems the inheritance of a given rule might be implicit, it might not feature inheritance or inheritance is controlled through a distinct field. In many systems if an explicit rule is not matched the system will have a implicit deny to match the request.

The *inheritance* describes whether this particular rule should be inherited and how this should occur. This permits control over whether a rule applies to child items stored within the entity or only to the target entity. Inheritance rules only apply to container items — e.g. items that can contain other items. Non-container items do not have any inheritance ability. Inheritance rules may apply to both the entity itself and its child items in addition to not inheriting (only applying to the entity) or inheriting to just its children. In some systems inheritance of a permission rule is explicitly stated however in other system inheritance may be implicitly followed through a hierarchy unless explicitly overridden.

1.3.5 Capabilities

Capabilities or Privileges are system level access controls. They don't necessarily provide fine grained access control to items but provide access control to generic items. Examples of capabilities for Linux is the ability to permit an application to bind to a privileged port without requiring the full root credentials (e.g. a port under 1024). Microsoft calls the equivalent of capabilities in Windows "privileges" and defines various privileges such as the ability to take ownership of a file, impersonate a user or load drivers.

It should be noted that the Linux implementation of capabilities is best described as stalled and is implemented differently to the way Windows handles its privilege system. Windows privileges are granted to a user or a group of users and can be exercised at any point by these users.

A more detailed comparative discussion on privileges and capabilities is covered in 5.3.

It should be noted that Windows privileges are distinct from the privileges used in Structured Query Language (SQL). SQL privileges are analogous to actions as described in 1.3.4.

1.3.6 Roles

Roles are collections of permissions that can then be assigned to users or groups of users. Role based access control, covered in 2.3, can be emulated through different means. Roles may or may not have users. Roles are distinguished from groups as a group's primary purpose is to contain users while the primary purpose of a role is for permissions to be assigned to it. In a role based system the entity is a role not a user or a group. In true role based systems, users and groups cannot be assigned permissions directly. Roles are only highly effective in situations where access control scenarios can be effectively classified. Lack of easy classification can lead to excessive proliferation of roles that can make them less than effective for management. So while roles can be useful in a number of situations, those situations aren't considered in great detail in this dissertation. Coverage of roles is limited within this dissertation with more detail in 2.3 and 3.6.5.

1.4 Summary of the Dissertation

In this chapter I have introduced a background of various systems and a quick overview of where access controls in various systems have come from and where this work fits in. It provided a short introduction to access controls in file systems, in web systems and databases as well as access controls in general.

The second part includes an in depth literature survey and also some personal research into the behaviour of various systems. The first chapter of part two handles a review of literature in the area of access control whilst covering in itself a portion of access control history. The second chapter handles a more historic review of existing implementation and the choices that have been made within them. It studies varies methods of implementing access control in file systems, database systems and web information systems. The third chapter covers scenarios of access control focusing on existing access control systems. The last chapter handles a detailed examination of present file system access control models including exposing interesting behaviours in file systems.

The third part introduces MDFS — a metadata file system developed for research. The file system is scoped out in the first chapter with an examination of the purpose of the system. The second chapter of the part lays out the specification of the file system. The third chapter handles the various design aspects of the system

before the final chapter details the implementation of the system and examples of the system at work.

The final part concludes the dissertation and introduces some access control scenarios with possible solutions. Areas of future work and future access control are introduced. This part also includes a glossary of terms followed by the bibliography.

Part II

Review

Chapter 2

Literature Survey

In this chapter we examine literature relating to access control, in particular some basic theories pertaining to access control. These theories, such as access control lists and role based access control, are implemented in the access control systems examined later in this chapter.

2.1 Access Control Basics

Access controls at a basic level define actions that might be permitted or restricted. Access control can refer to any type of action that a user might or might not be able to complete over an object. Commonly file systems have two basic access controls: the ability to read an object and the ability to write an object. UNIX operating systems implement a third permission called "execute" which covers which files the kernel will load code to execute with [60]. At the most basic level a set of permissions is granted to the owner of the object and to all other identities within the system.

Access control systems take several different forms. The typical UNIX style [60] is boolean: it works by granting permissions explicitly and anything not granted is a denied. Other systems, such as NTFS, have the tristate permissions: they permit explicitly granting permissions, explicitly denying permissions and inheriting permissions (default state). NTFS introduces the concept of inheritance for permissions as well so if a permission isn't specified at a lower level of the directory tree, the directory tree is traversed from the leaf directory through the ancestors to the root directory until an explicit permission is found. If no permission is found then a deny is used in this case. Moodle not only has permit, deny and inherit but it also introduces the concept of prohibit. Prohibit prevents the user from executing that action from that level down the tree in the context system. Prohibit will also prevent a permission from being granted at a lower level when this would otherwise be possible.

2.2 Access Control Lists

Access Control Lists provide a list of defined controls (such as read, write, execute) and providing the ability the apply rules for these permissions to a list of users. This is distinct from the UNIX model where access rights in relation to a file are defined for a single user (the owner) and a single group (the identity of which may vary). ACL based systems permit much more fine grained control of permissions by permitting owners of objects to grant multiple groups or users varied level of access to the files. In its simplest form, an Access Control List could be considered

similar to the list that a bouncer of a club has: users on the access list are granted the permission to enact the action of entering the club.

2.3 Role-Based Access Control Models

Role Based Access Control Models introduced by Sandhu et. al. [91] provide a model for provision permissions to a 'role' within a given system. Roles permit a user to assign a set of predefined permissions to an entity which can then be associated with a set of users. A key difference between groups and roles is that with RBAC the permissions applied to roles are known in a central location or database. By contrast, a typical file system where the permissions granted to individual groups can be spread across the file system. The decentralisation of access control rights in a file system can make auditing hard to determine exactly which permissions a particular user or group may have. Thus a strong advantage of role based access control is that the permissions of a user can be added or removed easily from a single location which controls the access.

RBAC also solves the problem of organisational permissions change. If a person starts in a new position they can be assigned the role of their position and have the appropriate permissions applied to them. This also works when permissions are added or revoked to a role as it then applies to all users who are associated with that role. Roles are also presented in a manner in which they can form a tree to provide inheritance for permissions.

A good way to think of a true RBAC is that permissions are always applied to groups of users, even if its group with one or even zero users in it, and never directly to the users. This means the exact permissions of a user is easily auditable by examining their groups and thus the permissions granted to the groups.

Some of the concepts introduced in RBAC are interesting and perhaps could be modelled in part by MDFS, however, the objectives of Role Based Access Control appear to be largely unrelated to the use of metadata in connection with access control. Due to the design of the file system as a database, MDFS could implement roles if permissions can be assigned to abstract entities and then these permissions are applied to related entities (e.g. a 'role' entity could be created with permissions assigned to it and a 'member' relationship to other objects which could then use these permissions.

Role Base Access Control has developed into a few filesystems and can be seen as the basis of others. At the low end of the scale, if we consider that RBAC is effectively groups of users with permissions attached to the groups, one could considered that RBAC style permissions are available in NTFS if the permissions of users were removed. Bohra et. al. [63] have applied RBAC to a standard NFS mount using a file system proxy that they have developed. This system in this situation maps users back to active roles within the system.

2.4 Rule Set Based Access Control

Rule Set Based Access Control [14] (RSBAC), is an access control methodology for the Linux kernel. RSBAC provides for mandatory access control (MAC), access control lists (ACL) and role compatibility (RC) security models. RSBAC appears to also work with the Linux capabilities system as well. This is considered an extension of Generalized Framework for Access Control [71] which proposes rule set modelling. Most of this work is focused around the UNIX security model requirements however it is obvious that it is applicable elsewhere.

The GFAC model exposed in Figure 2.1 depicts the flow of access between

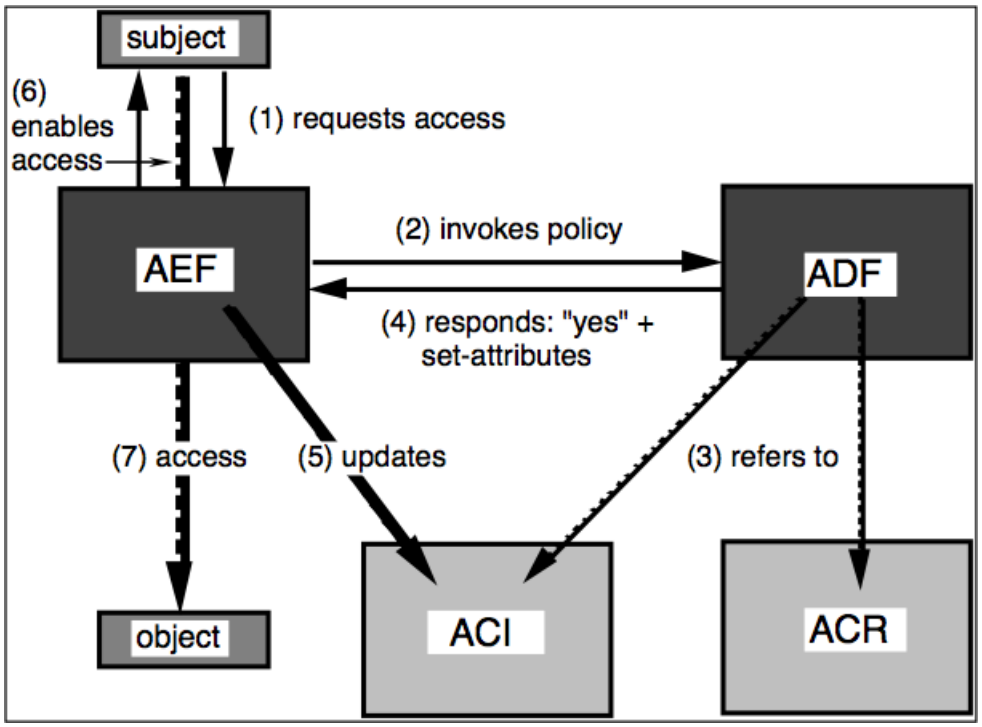

Figure 2.1: Overview of the Generalized Framework for Access Control [71]

different aspcts. The access enforcement facility (AEF) which controls access to an object by a subject and can update attributes of a subject. The access decision facility (ADF) which determines if a particular subject can have access to a given object. The access control information (ACI) stores attributes that the ADF uses to determine if access should be granted, these are about either the subject or the object. The access control rules (ACR) are the rules that the ADF makes its decision with the input of the ACI information.

Olson and Abrams describe the main aspects of this system in an abstract manner [79]. Information and attributes about the subjects and objects (stored in the ACI) are combined with the rules (ACR) and which determines and enforces access (AEF). These terms are define in their work as:

- Information: attributes (characteristics or properties) of subjects and objects, and context (additional information used in the access control decision making, such as time of day).
- Rules: regulating principles that implement the access control policy by adjudicating access requests by subjects to objects.
- Authority: the set of agents who determine the rules and specify and assign values to attributes and context; multiple authorities may exist and have different spans of control.

It is apparent that rule based access control and the Olson and Abrams "Generic Framework for Access Control" has played a part in later work. Other concepts introduced in their paper include:

- A discussion on policy choices around "Confidentiality vs. Integrity vs. Availability" and "Maximized Sharing vs. Least Privilege", "Granularity of Controls" relating to Mandatory Access Control (MAC) and Discretionary Access Control (DAC).

- There is discussion over controls relating to attributes of a user which is based on in part their role but also other attributes of the person. This includes their employment state (e.g. no contractors) or perhaps their nationality (e.g. no foreign nationals).
- There is discussion around using different attributes not relating to either the subject or the object to make a decision, entitled "external conditions". Examples provided include the time, location and even some external state (e.g. military readiness).

2.5 Relation Based Access Control

Relation Based Access Control (RelBAC) is an access control methodology which utilises the relationships between entities to form access control rules [67]. An ER diagram of the model, shown in Figure 2.2, shows a subject—permission—object relationship. RelBAC makes use of description logic to express the concepts in a formal manner. RelBAC is heavily guided by work in the area of web information systems.

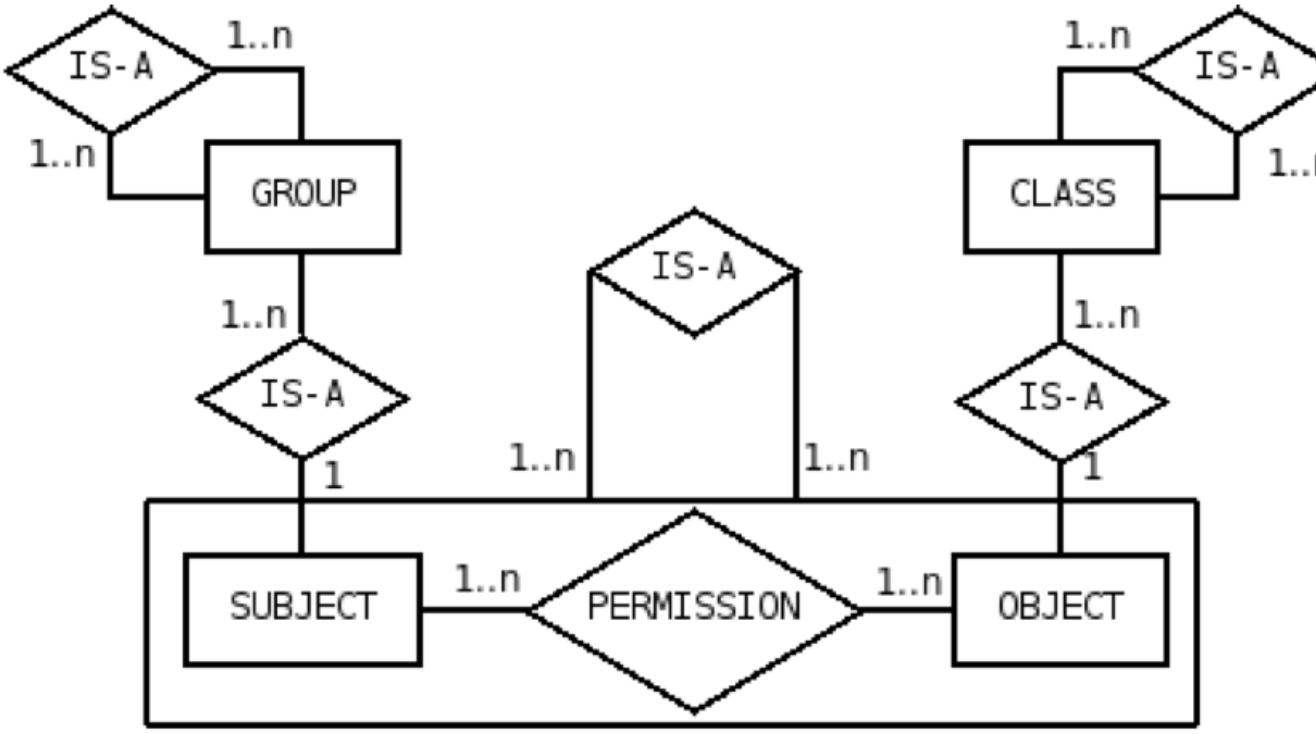

Figure 2.2: The ER Diagram of the RelBAC Model [67]

Chapter 3

Historical Review

3.1 Introduction

In this chapter we examine an implementation history of access control across various systems. As access control has evolved, various stages are examinable through looking at the systems that were implemented. While the literature review focused on current literature in file systems and concepts that are still evolving in addition to more historic references, this chapter will cover reports primarily from software vendors about what they have implemented or defined standards.

A historical review is important because it allows the charting of the historical development of access control. As new access control systems are devised they are developed from the understandings of previous systems. This is grouped into a selection of major areas of study from database systems to web based systems with each section containing material from the relevant area of study. A metadata file system may be thought of as a file system in which attributes associated with files are also stored, and may be accessed, via a database. Web applications also require complex access control mechanisms which could readily be implemented as a meta-data file system.

Access control is implemented differently depending on perspective. Take for instance databases where access control is applied to the server or schema level but typically not at the row level. Normally with a database if you can select from a table then you can select from all rows of the table: there aren't any row level restrictions. File system's work on a much lower level with permissions on the actual instance of the file not on the actual file system itself or even the directory. As such file system's are concerned more about permissions of the instance as opposed to the database which is concerned more with schema level permissions in general.

Metadata file systems have some characteristics of traditional file systems and some of database systems. Further to this, a metadata file system mimics functionality that is available in modern web based permission systems. Web based systems exist within the bounds of a database system but also have their own model of permissions that are implemented by the web application themselves.

Web based systems can implement very simple permissions — there might only be two levels: a user who is registered can do everything and a user who isn't registered, an anonymous user, can do nothing. Web based systems can also scale up to be similar to full role based systems. Oracle's PeopleSoft product is an example of this where the web interface is controlled by an extensive set of definable roles to which a user can be assigned. These roles can also be granted based on the users status. For example a manager who supervises staff might be granted the role to be able to view their staff's leave requests and current leave balances. This in part

involves relational awareness: a manager should only be able to view the leave of their own staff and no others. So in this case the action is to view a subordinates details.

Individual systems have unique features which make them behave in distinct ways around their access control systems, of which the execute permission is a good example. In file systems it is used to restrict which files might be executed by the kernel, either binary or script (although script applications can usually be executed individually by their interpreter presuming it is executable). Any user might set an executable bit on a Linux operating system which makes it limited for use access control in general. When we look at the similar execute permission in database systems, the access control system is capable of more detailed control: a particular user might be able to create an executable routine but may not have the permissions to execute them. Additionally a user might be able to execute a given set of routines, they are just unable to execute a new routine they have written. So while in the Linux operating system, the owner of the object can control which actions are accessible, in a database system the user has to be given the ability to grant a particular action before the user can permit a given action (e.g. owning an object doesn't necessarily permit control of its permissions).

3.2 Identity Management

As operating systems these days almost entirely support the concept of multiple users, this support needs to be reflected in the implementation of file systems. Further complicating this task is the advent of remote file systems which means that a users identity might not even be stored locally to the file system. As most operating systems define users, we need to identify a user to a given object within the file system. While metadata file systems might change the way users are identified and stored within the system, we will not examine this in detail and accept that the user can be identified and verified external to the problems presented.

3.2.1 Local Identification

The first step from a single user system is to a locally identified multiple user system. In UNIX based systems, users are stored in the "/etc/passwd" file with their username, user identifier, their group. User authentication is handled by a privileged process accepting a username and password, examining the password file (or a shadow password file for Linux systems [22]), validating the credentials and then creating the user's shell process in a new environment with their user ID set.

At this point the file system is responsible for storing the owner user ID of files and validate them against the user making the request (the user and group ID of the user requesting the operation is available to the file system operation). For example the FUSE high level API provides the "fuse_get_context" function which returns a struct containing the requesting user's ID and group ID. Because of this the basic UNIX based inode structure uses user ID and group ID to handle the owner and stores this on disk [60].

3.2.2 Pluggable Authentication Modules

Pluggable Authentication Modules [90], PAM, is a Linux technology which abstracts various identity related tasks such as session set up, password updating and validation, authorisation and account management. PAM provides four major modules to handle authentication management, account management, session management and password management. PAM abstracts this away from applications that use its

interfaces so that they don't have to worry about how the particular task is handled but that an appropriate result is obtained.

Authentication management handles identifying that a user is who they claim to be by validating their credentials. Examples of this could be the standard UNIX passwd files, Kerberos or any other method that can validate the users credentials. Account management handles account related restrictions such as if the user is blocked from logging in, if the account is perhaps expired or if there are limitations upon when the account can log in. Session management provides the ability to set up items in the session or do session related accounting, for example the duration for which the user was logged in. Password management provides a framework to update the users password.

Each of these interfaces are supported by service providers that handle the specifics for each individual system, for example the UNIX passwd password management module would handle updating the /etc/passwd or shadow password files while a Kerberos password management module would update the password on the Kerberos servers.

3.2.3 Distributed Identification

When dealing with distributed systems, local identification becomes impossible to use with out some form of distributed identification. There is nothing enforcing that user 501 on system A is the same as user 501 on system B - so some form of alternate centrally managed identification scheme.

As Gagne et. al. introduce in Operating System Concepts [94], there have been solutions to that with examples like Yellow Pages (later network information service, NIS) from Sun Microsystems and various LDAP [95] compatible directory services such as Microsoft's Active Directory system, Apple's OpenDirectory and the open source OpenLDAP system. These provide basic features such as mapping a username to a uniquely identifiable user identity, either a simple integer like UNIX or a more complex security identifier which Microsoft uses to identify users [87].

Prior to Windows 2000, Windows domain distributed identification was provided by "Windows NT Directory Services" which relied upon the primary domain controller/backup domain controller design. Active Directory, introduced in Windows 2000, was the first Microsoft directory services product to feature LDAP support [77].

3.2.4 Decentralised Identification

Decentralised identification is a newer scheme introduced primarily by OpenID [80]. OpenID is a method of identifying a person as owning a particular URI, typically their blog, home page or some other uniquely identifying URI. OpenID is supported by various systems such as Atlassian Crowd [49] and Joomla!.

3.3 Database Systems

Database systems have implemented various levels of permissions for controlling user's access to their data. This section will cover permissions implemented in database systems themselves while later sections will examine permissions stored in database systems however implemented in a third party application, such as a web application.

3.3.1 Basic Permissions

At its core SQL grants, to users, a list of privileges and also, optionally, the privilege to grant these privileges to other users. The privileges might be granted at various levels such as for the account level or at the relation level [65]. Relation level permissions grant the ability to select, modify and remove data from the system's existing data structures within a schema and account level permissions grant the ability to create new schemas within a database or new tables in addition to all of the privileges that a relation level permission grants. Some systems, such as MySQL, have the option to grant permissions on schemas as well as on the account level or relation level.

Actions in SQL systems are more complete when compared to the actions normally offered by most file systems. Actions in SQL also cater for the domain specific issues that are involved with databases such as items like stored procedures. Stored procedures are executable applications that can be created within the database and run like a normal application. The ability to create something that is executable is something that can be configured as opposed to the Windows or UNIX world where an application can be set as executable by the owner of the object. The difference with the database is that the permission is something that can be limited by the administrator. Databases also support delegation of access control so a action can be granted to a user with the grant option and they can then grant the same action to another user of the system.

MySQL, as detailed in Table 3.1, actually has three actions available with regards to stored procedures: ALTER ROUTINE, CREATE ROUTINE and EXECUTE. So a user may be able to create a routine but may not be able to execute it or alter it after it has been created. Depending on the database server there is the possibility to grant the particular action only on a given routine — so a user might be able to execute all stored procedures in a schema or only be able to execute or alter specific stored procedures within the system. They can also utilise the GRANT action to extend or limit the control of the system from there. This differs from the UNIX access control model (see 3.4.1) where if a person can create a file they can typically change it to become executable.

3.4 File system Permissions

File system permissions come in a variety of flavours, from basic systems that have absolutely no restrictions on access rights to the UNIX style file systems. UNIX style file systems have permissions based on the owner of the file (or group member) but limited ability to do fine grained permissions (either in their targets or their permissions). This is then in contrast to full blown ACL file systems which permit extensive lists of controllable actions, users/groups and the ability to apply complex inheritance structures for the resultant rules.

3.4.1 UNIX Permissions

UNIX file access permissions limits access to files based on three classes of users: those who own the file, those whom are members of the group which owns the file and every other user. Each of these three collections of users has permissions for reading, writing or executing. As noted in Bach [60], directories have the execute permission replaced with the right to search or traverse a directory for a filename. This is represented in its octal bit mask form (e.g. 777) or utilising characters (e.g. rwxrwxrwx) where each set of 8 bits represents user, group and world permissions respectively. The owners of the file (user and group) are stored in their unique ID form not as their name.

Table 3.1: MySQL permissions [34]

- CREATE
- DROP
- GRANT OPTION
- REFERENCES
- ALTER
- DELETE
- INDEX
- INSERT
- SELECT
- UPDATE
- CREATE TEMPORARY TABLES
- LOCK TABLES
- CREATE VIEW
- SHOW VIEW
- ALTER ROUTINE
- CREATE ROUTINE
- EXECUTE
- FILE
- CREATE USER
- PROCESS
- RELOAD
- REPLICATION CLIENT
- REPLICATION SLAVE
- SHOW DATABASES
- SHUTDOWN
- SUPER
- ALL [PRIVILEGES]
- USAGE

UNIX also has a fourth octal field for permissions. This field is composed of the setuid bit, setgid bit and sticky bit. When set to files, the setuid and setgid bits cause the executable to run as the owner of the file and as the group respectively. This is typically used to run tools that require root privileges without having to grant the user root privilege. A simple example of this is "ping", which requires root privileges to send and listen for control packets on the network interface directly [51]. The setuid bit when set to a directory does nothing under Linux [82] however when the setgid bit is set to a directory any new files created in this directory will have it's group set to the same as the parent directory. On Mac OS X, if the setuid bit is set on a directory, then the files created within that directory are owned by the user who owns the directory (if supported by the underlying file system) [19].

The sticky bit is the final part of the mask. This when applied to files is ignored under Linux, however when the sticky bit is set on a directory, files in that directory may be unlinked or renamed only by root or their owner. Without the sticky bit, anyone able to write to the directory can delete or rename files. The sticky bit is commonly found on directories, such as /tmp, that are world-writable. [72] This behaviour is consistent with Mac OS X [57].

The sticky bit has different behaviours on different platforms, the Wikipedia article has a comparison citing man pages of various operating systems [52]. Linux, FreeBSD, OpenBSD and Mac OS X appear to ignore the bit. Operating systems such as AIX 5.2, Solaris 10, HP-UX, IRIX and SCO UnixWare appear to use the bit on executable files to retain the program text in swap or memory as a performance boost.

3.4.2 POSIX ACL on Linux

In POSIX Access Control Lists on Linux [78], Gruenbacher methodically studies the implementation of various ACL implementations focusing on Linux. Gruenbacher covers detailing how ACL's work according to the POSIX standard, how they have actually been implemented and also examples of compatibility across networks with NFS and Samba. Performance is also examined with in one case a 500 times increment in access times recorded for a file system with ACL as opposed to the same system without ACL's. The paper documents various other performance comparisons with nested directories and copying files.

3.4.3 FAT Permissions

While the FAT file system [29] is a single user file system (e.g. there is no provision to store owner identifiers for files), the FAT file system through the "read only" attribute does introduce a primitive form of permissions. As the system doesn't have any concept of owners, any users could remove the read only attribute making it ineffective at prevent malicious activity but useful for preventing inadvertent mistakes. Other attributes that FAT supports are "archive", "system" and "hidden".

3.4.4 ext2+ Permissions

ext2, or the second extended file system, not only provides the permissions that are common to UNIX based file systems (read, write and execute for user owner, group owner and world), but ext2 also introduces the concept of extended attributes [92] as listed in Table 3.4.4.

As these are file system specific attributes, "chattr" tool from the e2fsprogs package is required to set these attributes and the normal Linux "ls" command will not list them (the "lsattr" tool from the same package however will).

- a - Append Only: The system should only allow opening of this file for appending and should not allow any process to overwrite or truncate it. In the case of a directory, processes may create or modify files in the directory but not delete them.
- i - Immutable: The system should disallow all changes to this file. In the case of a directory, processes may modify files that already exist in the directory but may neither create nor delete files.

Table 3.2: ext2 Extended Attributes

3.4.5 NTFS Permissions

The "New Technology File System", commonly referred to as "NTFS", provides a richer permissions model than UNIX does by default, somewhat similar to the power that the POSIX ACL system offers. Unfortunately as Russon et. al. notes [87], there is very little official documentation on NTFS from Microsoft so much of their research is collations from working through the limited documentation that Microsoft does provide. The MSDN fortunately documents "Security Features for File Systems" [42] which has a section on the Access Mask used by the system to determine permissions.

Actions

The access mask in Figure 3.1 appears to be split into sections: generic rights, standard rights and specific rights. Generic rights mimic the UNIX read, write, and execute permissions and also feature an "all" which encompass all of the previous three permissions. Standard rights are applicable to all objects within the file system and control if the object can be deleted, if there is the right to read the security permissions, write the security permissions or alter the owner of the object. The last section, specific permissions, are permission delineated by whether the object is a file or a directory.

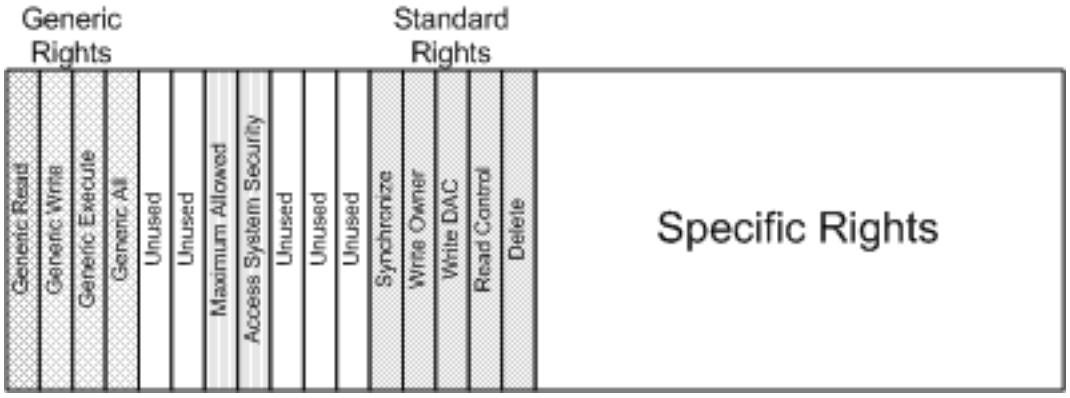

Figure 3.1: Actions under a NTFS file system

Microsoft lists the actions show in Figure 3.1 as [36]: generic rights (Table 3.3), standard rights (Table 3.4) and specific rights. Specific rights are classified as file specific (Table 3.5) and directory specific (Table 3.6).

NTFS utilises three potential modes for a permissions: permit, deny or inherit. If no permission is inherited along the tree then this defaults to a deny. The calculation

Table 3.3: NTFS Generic Rights (analogous to UNIX "rwx")

- GENERIC_READ — the right to read the information maintained by the object.
- GENERIC_WRITE — the right to write the information maintained by the object.
- GENERIC_EXECUTE — the right to execute or alternatively look into the object.
- GENERIC_ALL — the right to read, write, and execute the object.

Table 3.4: NTFS Standard Rights (applies to all objects):

- DELETE — the right to delete the particular object.
- READ_CONTROL — the right to read the control (security) information for the object.
- WRITE_DAC — the right to modify the control (security) information for the object.
- WRITE_OWNER — the right to modify the owner SID of the object. Recall that owners always have the right to modify the object.
- SYNCHRONIZE — the right to wait on the given object (assuming that this is a valid concept for the object).

Table 3.5: NTFS File Specific rights

- FILE_READ_DATA — the right to read data from the given file.
- FILE_WRITE_DATA — the right to write data to the given file (within the existing range of the file).
- FILE_APPEND_DATA — the right to extend the given file.
- FILE_READ_EA — the right to read the extended attributes of the file.
- FILE_WRITE_EA — the right to modify the extended attributes of the file.
- FILE_EXECUTE — the right to locally execute the given file. Executing a file stored on a remote share requires read permission, since the file is read from the server, but executed on the client.
- FILE_READ_ATTRIBUTES — the right to read the file's attribute information.
- FILE_WRITE_ATTRIBUTES — the right to modify the file's attribute information.

Table 3.6: NTFS Directory Specific rights

- FILE_LIST_DIRECTORY — the right to list the contents of the directory.
- FILE_ADD_FILE — the right to create a new file within the directory.
- FILE_ADD_SUBDIRECTORY — the right to create a new directory (subdirectory) within the directory.
- FILE_READ_EA — the right to read the extended attributes of the given directory.
- FILE_WRITE_EA — the right to write the extended attributes of the given directory.
- FILE_TRAVERSE — the right to access objects within the directory. The FILE_TRAVERSE access right is different than the FILE_LIST_DIRECTORY access right. Holding the FILE_LIST_DIRECTORY access right allows an entity to obtain a list of the contents of a directory, while the FILE_TRAVERSE access right gives an entity the right to access the object. A caller without the FILE_LIST_DIRECTORY access right could open a file that it knew already existed, but would not be able to obtain a list of the contents of the directory.
- FILE_DELETE_CHILD — the right to delete a file or directory within the current directory.
- FILE_READ_ATTRIBUTES — the right to read a directory's attribute information.
- FILE_WRITE_ATTRIBUTES — the right to modify a directory's attribute information.

of these permissions appears to be through backwards traversal from the object in question up the tree until an explicit permit or explicit deny is reached. This means that an explicit permission further up the tree will be ignored in place of an explicit permission further down the tree. At the same level a deny will override a permit, so if the user is in a group with an explicit deny for the object but is granted permission under their user account they still won't have permission as the deny will overrule the permit.

NTFS also features all of the attributes available in the FAT file system, including read-only, system, hidden and archive. Additional features appear to be the ability to control if something is indexed for search operations, if a file is compressed or a file is to be encrypted.

Privileges

Windows as an operating system also provides the following "privileges" that a user might obtain, analogous to Linux "capabilities". File system related privileges include [41]:

- SeBackupPrivilege — allows file content retrieval, even if the security descriptor on the file might not grant such access. A caller with SeBackupPrivilege enabled obviates the need for any ACL-based security check.
- SeRestorePrivilege — allows file content modification, even if the security descriptor on the file might not grant such access. This function can also be used to change the owner and protection.
- SeChangeNotifyPrivilege — allows traverse right. This privilege is an important optimization in Windows, since the cost of performing a security check on every single directory in a path is obviated by holding this privilege. This privilege also causes the system to skip all traversal access checks. It is enabled by default for all users. This privileges also provides the ability to register for change notifications on an object (e.g. a directory to update the file listing when a new file is added). It is peculiarly a privilege that controls "two unrelated powers" [86].
- SeManageVolumePrivilege — allows specific volume-level management operations, such as lock volume, defragmenting, volume dismount, and setting valid data length on Windows XP and later. Note that this particular privilege is explicitly enforced by a file system driver primarily based upon FSCTL operations. In this case, the file system makes a policy decision to enforce this privilege. The determination of whether or not this privilege is held by the caller is made by the security reference monitor as part of the normal privilege check.
- SeTakeOwnershipPrivilege — Take ownership of files or other objects. Required to take ownership of an object without being granted discretionary access. This privilege allows the owner value to be set only to those values that the holder may legitimately assign as the owner of an object.
- SeImpersonatePrivilege — Impersonate a client after authentication.
- SeCreateSymbolicLinkPrivilege — Create symbolic links.

Some of these privileges permit the process utilising the privilege to ignore any file system level security [3]. In all cases a privilege will take precedence over any other form of control on a file system object. A full list of privileges available within Windows is detailed on MSDN [50].

Parent ACE flag	**Effect on child ACL**
OBJECT_INHERIT_ACE only	Noncontainer child objects: Inherited as an effective ACE. Container child objects: Containers inherit an inherit-only ACE unless the NO_PROPAGATE_INHERIT_ACE bit flag is also set.
CONTAINER_INHERIT_ACE only	Noncontainer child objects: No effect on the child object. Container child objects: The child object inherits an effective ACE. The inherited ACE is inheritable unless the NO_PROPAGATE_INHERIT_ACE bit flag is also set.
CONTAINER_INHERIT_ACE and OBJECT_INHERIT_ACE	Noncontainer child objects: Inherited as an effective ACE. Container child objects: The child object inherits an effective ACE. The inherited ACE is inheritable unless the NO_PROPAGATE_INHERIT_ACE bit flag is also set.
No inheritance flags set	No effect on child container or noncontainer objects.

Table 3.7: NTFS Inheritance Flags [37]

Inheritance

NTFS also features a curious model for handling permissions. Called "discretionary access control list" (DACL), these are "access control list[s] that are controlled by the owner of an object and that specifies the access particular users or groups can have to the object" [38]. Any explicit access control entries (ACE) are placed in the DACL first with denies placed before allows and then inherited ACEs are added backwards up the tree to the root (e.g. the parent entries, then the grandparent entries) with the deny entries first followed by the allow entries [40] [39]. When evaluation time comes, the list is traversed until a match for the entry is found: if it is a deny then the request is rejected and if it is an allow then the permission is granted [35]. Prior to Windows 2000 or Windows NT 4.0 SP4 (e.g. Windows NT 4.0 SP3 and lower) there was no deny, only allow [9].

Permissions inheritance is also complicated. For container objects, there are four possible settings for permission (ACE) inheritance 3.7.

This is rather obscure looking, however the Table 3.8 provides a translation of the strings used in the Windows XP security dialogs. A copy of these strings in actions is depicted in Figure 3.2.

This means that permissions in NTFS can be a very complex matter to determine, far more complicated than any other file system examined.

A final aspect of access control entries in Windows/NTFS is that of auditing. Auditing flags are stored with the inheritance flags [44] and flag if an operation should be logged. There are two flags for successful access and unsuccessful access as noted in Table 3.10.

3.4.6 Novell File system Permissions

Novell have produced two major file systems, the first is Netware File system (NWFS) and the second called Novell Storage Services (NSS). Novell's file sys-

"Apply Onto" name	**Flags**	**ACE Flags**
"This folder only"	no value	No inheritance applies to ACE.
"This folder, subfolders, and files"	(OI), (CI)	All subordinate objects inherit this ACE, unless they are configured to block ACL inheritance altogether.
"This folder and subfolders"	(CI)	ACE propagates to subfolders of this container, but not to files within this container.
"This folder and files"	(OI)	ACE propagates to files within this container, but not to subfolders.
"Subfolders and files only"	(IO), (CI), (OI)	ACE does not apply to this container, but does propagate to both subfolders and files contained within.
"Subfolders only"	(IO), (CI)	ACE does not apply to this container, but propagates to subfolders. It does not propagate to contained files.
"Files only"	(IO), (OI)	ACE does not apply to this container, but propagates to the files it contains. Subfolders do not receive this ACE.
"Apply these permissions to objects and/or containers within this container only"	adds (NP)	This flag limits inheritance only to those sub-objects that are immediately subordinate to the current object.

Table 3.8: Access Inheritange Flags [15]; Legend for flags in Table 3.9

OI	Object Inherit
CI	Container Inherit
IO	Inherit Only
NP	No propagate

Table 3.9: Inheritance Modes

Audit flag	**Meaning**
FAILED_ACCESS_ACE_FLAG	Meaningful only in system-audit and system-audit object ACEs. The access mask specifies operations that should be logged when they fail.
SUCCESSFUL_ACCESS_ACE_FLAG	Meaningful only in system-audit and system-audit object ACEs. The access mask specifies operations that should be logged when they succeed.

Table 3.10: Audit Flags [43]

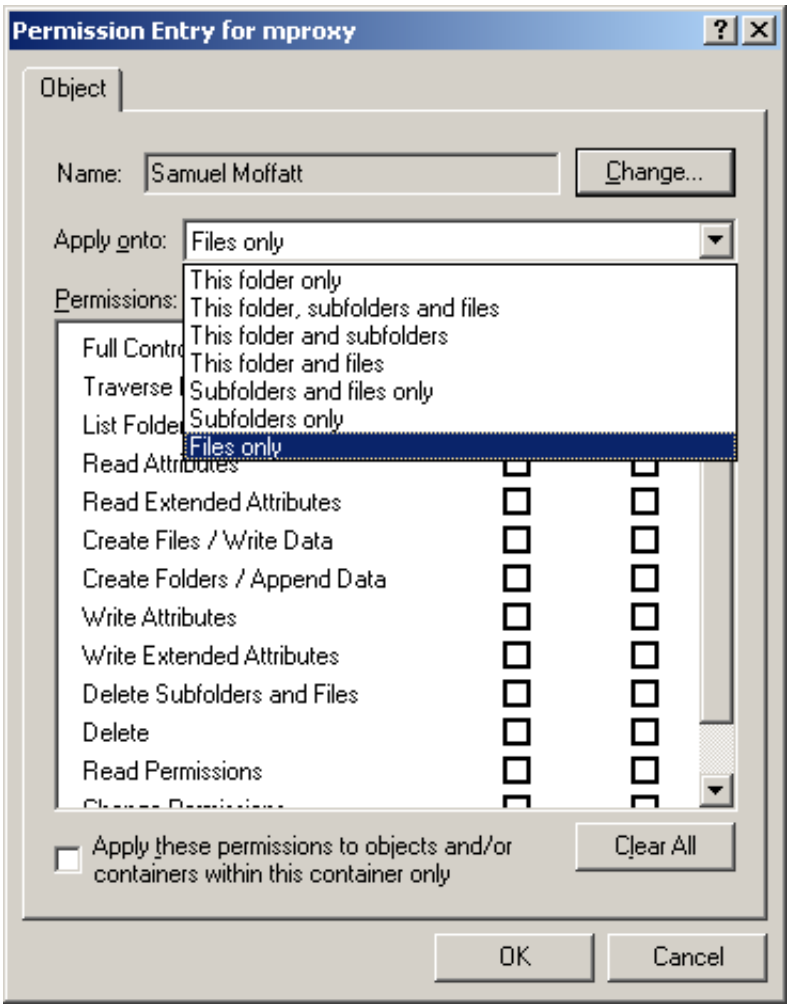

Figure 3.2: A screenshot of the advanced permissions dialog from Windows XP

tems feature extensive attributes that control the behaviour and access to a file or folder as well as extensive "trustee rights" which control user permissions. Some of those trustee rights are similar to actions that exist in other systems, such as read, write, access control, create and erase (aka delete). Some trustee rights are unique to NWFS such as supervisor and file scan. Table 3.11 is a list of "trustee rights" available to Novell administrators.

Interestingly Novell has a few distinct permissions introduced at this point. The "Supervisor" right is perhaps analogous to a "super user" style permission in UNIX based operating systems however the permission can be delegated on distinct items. "File scan" permits the user to see an item which can be distinct from being able to read or write the item (e.g. you may not be able to see the item but if you know its there you can operate on it) and the "access control" appears to grant the ability to manage an item and change the ACL for the file. The "file scan" right has been in part emulated by Microsoft with Access Based Enumeration (ABE) where items that the user doesn't have access to are not displayed, however it is still not identical [31].

Attributes behave like other file systems and apply equally to all users once they are set. Attributes can also impact of the behaviour of the file system. Table 3.13 is a list of NWFS and NSS file and directory attributes that are available that are

Trustee Right	Description	Default
Supervisor	Grants the trustee all rights to the directory or file and any subordinate items. The Supervisor right cannot be blocked with an IRF (Inherited Rights Filter) and cannot be revoked. Users who have this right can also grant other users any rights to the directory or file and can change its Inherited Rights Filter.	Off
Create	Grants the trustee the ability to create directories and files and salvage deleted files.	Off
Erase	Grants the trustee the ability to delete directories and files.	Off
File Scan	Grants the trustee the ability to view directory and file names in the file system structure, including the directory structure from that file to the root directory.	On
Modify	Grants the trustee the ability to rename directories and files, and change file attributes. Does not allow the user to modify the contents of the file.	Off
Read	Grants the trustee the ability to open and read files, and open, read, and execute applications.	On
Write	Grants the trustee the ability to open and modify (write to) an existing file.	Off
Access Control	Grants the trustee the ability to add and remove trustees for directories and files and modify their trustee assignments and inherited rights filters.	Off

Table 3.11: Novell Trustee Rights [11]

related to file access control.

Some attributes are very similar to attributes (and permissions) that exist in other systems such as hidden, read only, executable and archive. Some extra attributes control privileges available such as rename inhibit (prevents rename), delete inhibit (similar to the ext2 immutable bit) and copy inhibit (prevents copying a file).

3.4.7 HFS+

HFS+ is the extended version of Apple's Hierarchical File System, primarily used for its Mac OS X operating system but also used on Apple hardware devices such as the iPod [32]. As of Mac OS X Tiger [20], HFS+ supports ACLs that are mostly compatible with POSIX [26] [16]. HFS+ supports the following permissions [19].

Permissions applicable to all file system objects:

- delete — Delete the item.
- readattr — Read an objects basic attributes.
- writeattr — Write an object's basic attributes.
- readextattr — Read extended attributes.
- writeextattr — Write extended attributes.
- readsecurity — Read an object's extended security information (ACL).
- writesecurity — Write an object's security information (ownership, mode, ACL).

Attribute	Description	Applies to
Ci	Copy Inhibit prevents users from copying a file. This attribute overrides the trustee Read right and File Scan right.	Files only
Di	Delete Inhibit prevents users from deleting a directory or file. This attribute overrides the trustee Erase right. When it is enabled, no one, including the owner and network administrator, can delete the directory or file. A trustee with the Modify right must disable this right to allow the directory or file to be deleted.	Directories and files
Dm	Do Not Migrate prevents directories and files from being migrated from the server's server disk to another storage medium.	Directories and files
H	The Hidden attribute hides directories and files so they do not appear in a file manager or directory listing.	Directories and files
N	Normal indicates the Read/Write attribute is assigned and the Shareable attribute is not. This is the default attribute assignment for all new files.	Directories and files
P	Purge flags a directory or file to be erased from the system as soon as it is deleted. Purged directories and files cannot be recovered.	Directories and files
Ri	Rename Inhibit prevents the directory or file name from being modified.	Directories and files
Ro	Read Only prevents a file from being modified. This attribute automatically sets Delete Inhibit and Rename Inhibit.	Files only
Rw	Read/Write allows you to write to a file. All files are created with this attribute.	Files only
Sh	Shareable allows more than one user to access the file at the same time. This attribute is usually used with Read Only.	Files only
Sy	The System attribute hides the directory or file so it does not appear in a file manager or directory listing. System is normally used with operating system files, such as DOS system files.	Directories and files
X	The Execute attribute indicates program files such as .exe or .com.	Files only

Table 3.13: Selected Novell File and Directory Attributes [10]

- chown — Change an object's ownership

Permissions applicable only to directories:

- list — List entries.
- search — Look up files by name.
- add_file – Add a file.
- add_subdirectory — Add a subdirectory
- delete_child — Delete a contained object.

Permissions applicable only to non-directory file system objects:

- read — Open for read.
- write — Open for writing.
- append — Open for writing but only at the end of the file.
- execute — Execute the file as a script or program.

Within HFS+, a file might be deleted if the user has the permission to 'delete' on the file or the 'delete_child' permission on the containing folder.
HFS+ also features the following inheritance capabilities:

- file_inherit — Inherit to files.
- directory_inherit — Inherit to directories.
- limit_inherit — Entries inherited from the parent will have the directory_inherit flag removed on the permission so that it doesn't inherit to grandchildren.
- only_inherit — The entry is inherited by created items but not considered when processing the ACL.

HFS+'s canonical form for the ACL is the following:

- local deny
- local allow
- inherited deny
- inherited allow

Normally the chmod "+a" operation adds ACL's to the list in a canonical form, however the "+a#" mode allows for setting the exact location allowing non-canonical entries (e.g. potentially an overriding local allow ahead of the local deny instead of the canonical location for a local allow which is after a local deny).
HFS+'s behaviour is very similar to that of NTFS.

3.4.8 ZFS

The Zettabyte file system, more commonly referred to as "ZFS", is a new file system developed by Sun for its Solaris operating system.

ZFS supports the NFSv4 ACL model which is itself heavily based on the Windows ACL model implemented in NTFS. ZFS defines two ACL specification formats as introduced in [69]:

Syntax A

```
ACL_entry_type:Access_permissions/.../[:Inheritance_flags]:deny or allow
```

ACL_entry_type includes "owner", "group", or "everyone". For example:

```
group@:write_data/append_data/execute:deny
```

Syntax B

```
ACL_entry_type:ACL_entry_ID:Access_permissions/.../[:Inheritance_flags]:deny or allow
```

ACL_entry_type includes "user", or "group". ACL_entry_ID includes "user_name", or "group_name". For example:

```
user:samy:list_directory/read_data/execute:allow
```

Inheritance flags

- f : FILE_INHERIT
- d : DIRECTORY_INHERIT
- i : INHERIT_ONLY
- n : NO_PROPAGATE_INHERIT
- S : SUCCESSFUL_ACCESS_ACE_FLAG
- F : FAILED_ACCESS_ACE_FLAG

It should be noted that the last two flags, successful access and failed access are actually audit flags and distinctly resemble the NTFS flags of the same name introduced in Table 3.10.

3.5 Distributed File systems

Distributed file systems are file systems that work between distributed machines. While with this dissertation the majority of issues will be examined in the context of running a file system on a single local machine, some of aspects of distributed file systems are interesting and have been examined for reference.

A paper by Hitz et al [59] covers the work done by Network Appliance's with their file servers and examining how they handle interaction from Windows clients onto UNIX backed permission stores and vice versa. Merging permissions provides an interesting problem set: UNIX's permissions are far more restrictive than the expressive model offered by NTFS which allocates up to 64KB to store ACL's compared to the 32 bits used for UNIX permission storage. Berger [61] introduces the differences between the two operating system approaches: NFS comes from the UNIX perspective and handles mounting a common share point and the protocol handles file permissions from this point. CIFS from the Windows world handles mounting share points from the perspective of an individual user (the user is authenticated when the mount point is created).

3.5.1 CIFS

The Common Internet File System is derived from Microsoft's Server Messaging Block (SMB) protocol and provides the same level of security that is afforded by the standard Windows NTFS security model.

3.5.2 NFSv4 ACLs

NFS, Sun's Network File System, gained support in version 4 for ACL's to restrict access to various systems. According to IBM, NFSv4 ACLs are supported on ZFS, JFS2 and GPFS [69]. NFSv4's ACL model is strongly based on that provided by Microsoft with their permissions matching almost perfectly with that offered by NTFS.

3.6 Web Based Applications

Increasingly the internet is driven from a "read-only" situation into a more "read-/write" situation. As this has occurred the need to apply permissions to users who create content has increased with non-technical users now maintaining websites with the aid of content management systems such as Joomla! [48] and increasingly third party users generating content for web systems such as Mediawiki [28]. Social networks also play a part in this sphere as depending on the 'network' of a user can define what permissions that user has to view content (e.g. a third level friend may only be able to see your name and your link while a first level friend can see all of your details).

3.6.1 RelBAC

RelBAC or Relation Based Access Control is a paper by Giunchiglia et. al. (2008) which covers building relationships not based on predefined groupings, such as RBAC, but by inferring permissions based on the relationships between particular objects. While Giunchiglia places this in the context of Web 2.0 applications and the relationships defined in those systems, the ideas presented could equally and easily be applied within the context of MDFS and its relationship model.

3.6.2 phpGACL

phpGACL [23], or PHP Generic Access Control Lists, is an open source ACL library written in PHP using MySQL as a data store. phpGACL has a flexible permissions model with three main concepts: ACOs ("Access Control Objects"; controls), AROs ("Access Role Objects"; requestors) and AXOs ("Access eXtension Objects"; instances). So in this model an ACO represents tasks the user can do, such as view a file, delete a file or similar, the ARO defines objects requesting to do ACO actions, such as a user, and the AXO permits the ability for finer grain control of ACO's (e.g. access for an individual item as opposed to all items).

Without an AXO, phpGACL could be considered schema based permissions (similar to the way database permissions are applied) where as with the addition of AXO it becomes possible to limit requestors to particular controls of instances instead of just at the schema level.

phpGACL appears to follow a 'role' based model where roles are defined, permissions are granted to those roles and users are then associated with those roles and have the permissions granted to them.

3.6.3 Joomla! 1.6

Joomla! 1.6 introduces a new fine grained ACL library that was previously unavailable. Earlier versions of Joomla! (1.5, 1.0) used a modified and limited version of the phpGACL library to achieve its work. Joomla! 1.6 introduces a new ACL infrastructure with a lot of the common themes as phpGACL though different terminology.

Joomla! 1.6 introduces two access control methods to control actions within the system. These are:

- Discretionary Rules — These are discretionary ACL's that apply to specific items. They permit users or groups to be permitted or denied the ability to complete a given action. They can apply to individual items or containers (e.g. an article, a category, a component or the entire site).
- View Rules — These rules are a mix of roles but only apply to a single 'view' action. View rules are access levels which can be assigned to items and groups can be assigned to an access level. So the ability to view an item is calculated by retrieving a list of groups that is in the access level and checking if the user is in the group.

In effect Joomla! has two distinct access control models for controlling access: rules to control most actions and a set of rules to control view permissions. The view rules provide the ability to easily cache a large amount of data (e.g. the users access levels can be cached on login) that enable the checks to be very cheap. As Joomla! is a database driven applications a query can easily be constructed to easily exclude items that the user doesn't have the ability to view which moves the access control into the database which can optimise results based on its indexing.

3.6.4 Moodle

Moodle has a complex "roles and capabilities" [13] system that permits flexible management of access to privileges in a system. Moodle is a RBAC implementation that defines capabilities that can be assigned to roles within a 'context' that define individual permissions. Permissions for Moodle are:

- not set or inherit - has no impact on calculation
- allow — Allow the role to enact the capability within this context or any child contexts
- prevent — Prevent the role from enacting the capability within this context or any child contexts.
- prohibit — The prohibit is a special permission that prevents the capability in any context.

Contexts in Moodle are equivalent to both container and objects - a context can have permissions to applied to them and they may also contain other contexts that inherit permissions or define their own permissions.

Moodle has a curious method for determining permissions [7]: it looks for a prohibit permission in any role in any of the contexts and if there is it stops and deny's access. If there are no prohibits, Moodle then examines all roles and calculates how many allow's and prevent's there are: if there are more allow's than prevent's then the permission is granted and if there are more prevent's than allow's the permission is denied. In the case of a tie the system defaults to denying access.

3.6.5 Sharepoint

Sharepoint is Microsoft's collaboration platform and forms a part of the Microsoft Office suite in the enterprise. Sharepoint offers shared calendars, lists, groups and wiki's. It provides the ability to store Office documents and work with InfoPath files [6].

Sharepoint Services 3.0 has five default "permission levels" that are available by default. These are Full Control, Design, Contribute, Read and Limited Access [12]. The Full Control and Limited Access permission levels cannot be modified or deleted. Limited Access is special permission level that cannot be assigned but it is automatically assigned when a permission level is granted to a child item higher than their current level. The Microsoft documentation cites the following example:

> For example, if you grant users access to an item in a list and they do not have access to the list itself, Windows SharePoint Services 3.0 automatically grants them Limited Access on the list, and also the site, if needed.

These permission levels serve as "roles" which can be assigned to users and groups. In granting permissions, a user can be allocated to a group or they can have a role manually assigned to them. By default each Sharepoint Site has three groups created: owners, members and visitors [2]. In the default configuration the owners group is granted the full control role, the members group is granted the contribute role and the visitors group is granted read.

Permission levels are by default inherited from a parent item to a child item however a child item may define "unique permissions" [1]. These unique permissions are initially a copy of the permissions that would ordinarily be inherited and are then customised. At this point changes to the original parent permissions are not applied to the unique permissions on that item. Also children of this item then inherit the unique permissions of the parent (as opposed to the grandparent) and so do their children until another item has unique permission levels set. It is my understanding that by "unique permissions", Microsoft means that the permission levels are customised.

3.6.6 XACML

XACML, or eXtensible Access Control Markup Language, is an XML based method for expressing access control rules. XACML has primitives for determining subjects, resources and actions which can be combined to form target matching for rules, policies and policy sets. XACML on its own doesn't enforce any restrictions but is only a format for describing permissions. XACML requires various infrastructure to evaluate its rules and apply them. The XACML model with policy enforcement points (PEP), policy decision points (PDP), policy information points (PIP), policy administration point (PAP) match the model that RSBAC and GFAC introduce in Section 2.4. Table 3.14 lists the equivalent terminology which seem to change "access" to "policy" and "facility" to "point".

The primary issue with XACML is the excessive verbosity in which rulesets are expressed via XML. XACML attempts to be in some respects too prescriptive and encompass many aspects of access control while making it hard to express existing access control systems (see Section 5.2 for a comparison of XACML and NTFS).

XACML appears to be included as a part of SAML however there are few case studies available that demonstrate the practical implementation of XACML.

XACML Term	**GFAC Term**
PAP	ACR
PEP	AEF
PIP	AIF
PDP	ADF

Table 3.14: Translation of XACML terms to GFAC terms

3.7 Conclusions

Looking over everything there seems to be some common categories for systems:

- Role based systems
- Discretionary ACL systems
- Permission inheritance
- Permissions with a boolean state (set or not set)
- Permissions with a triadic state (grant, deny, unset)

Role based systems, such as Moodle, work well when dealing with a limited set of roles and a large number of users. Some systems support explicitly the concept of roles and enforce that permissions can only be applied to a role not to an individual user however other systems may permit emulation of this functionality. For example Windows permits permissions being granted to either users or groups so if groups are only ever used to be assigned permissions and then users added to those groups the system is effectively a role based system.

Discretionary Access Control List (DACL) systems permit individual users or groups to be granted or denied actions to individual items which can optionally be inherited. DACL systems appear to be common and permit a level of control. DACL appears to be the dominant form of ACL for a lot of items as it permits the user to control who can access which files. UNIX and Linux by default offer primitive DACL with their default options (user, group and everyone) while POSIX ACL expands this to be a lot more flexible. Windows and NTFS appears to have the most extensive ACL system for file systems offering a wide range of permissions and inheritance capabilities that is emulated by other systems like HFS+, NFSv4 ACLs and ZFS.

Inheritance of permissions appears to be primarily lead by the NTFS implementation which provides various controls for inheritance and application of ACLs. Various other systems have their own forms of inheritance though this is usually simple. Inheritance evaluation rules vary widely between the various systems and the use of Mandatory Access Control (MAC) style rules (e.g. Window's SeChangeNotifyPrivilege) further complicate the situation.

MAC systems are not particularly strong with the exception of privileges and capabilities exhibited by Windows and Linux respectively. MAC isn't particularly visible in the systems identified throughout the literature survey however this is perhaps due to MAC being more relevant to defence organisations which have much more rigorous controls on information transmission and access control.

Permissions come with either boolean state (set or unset, like UNIX, POSIX, POSIX ACL and early Windows implementations) or the triadic variety (grant, deny, unset; like Windows and derivatives). The boolean state permits the ability to grant users access but not necessarily to easily remove or deny access to them. The triadic permissions have the advantage that they can deny access but this then

causes complications in rule evaluation algorithms which results in divergence of algorithms and choices. The Windows model would appear to not permit directory traversal if the parents don't allow it however the SeChangeNotifyPrivilege allows this check to be circumvented for performance reasons. It appears that the combination of flexible access control and triadic permission states increases the amount of processing required to make an access control decision.

Chapter 4

Scenarios

This chapter describes some scenarios around permissions in file systems. It will cover some current scenarios available as well as introduce some more advanced concepts.

4.1 Basic Concepts

Before moving into the scenarios there are some basic concepts that are used within the system.

4.1.1 Entities

Entities are parts of the system that are capable of completing various actions. An entity is most familiar as a user who completes actions. Entities can also be groups of users or roles. In which case those users who identify as being a member of that particular group or having that particular role are thus capable. Entities may also form the part of services or agents which may also wish to act autonomously or even on the behalf of other entities (e.g. a user). Entities usually play the role of subject in the sentences in which they appear.

4.1.2 Actions

Actions are tasks that might be completed on an object by an entity. Actions are completed by entities upon objects and are verbs. Examples of actions include reading, writing, deleting, appending and creating. Actions might also describe more complicated processes such as the ability to authenticate to a particular system. Actions usually play the role of verbs, or predicates, in the sentences in which they appear.

4.1.3 Objects

An object is a resource of that is being protected. Objects are typically thought to be files and folders however objects may also be users, groups, role or any other resource that can be controlled or secured. Within UNIX and its derivatives (Linux/Mac OS X) an additional amount of items are represented as files such as pipes, sockets or devices (block special file or character special file). This permits access control of these particular items. Windows has a slightly more comprehensive model for securable objects which also includes processes and threads, printers and directory service objects (e.g. users, computers, groups, etc). Objects usually play the role of objects, in the grammatical sense, in the sentences in which they appear.

4.1.4 Relationships

Objects within the system can be connected to each other through relationships. Relationships are set up to be directional and should be read from left to right as depicted in Figure 4.1. In a fully relational file system, entities can also be objects themselves. In such a system it could codify relations as objects which makes applying and manipulating them subject to the standard rules of the file system. This is an extension of the UNIX concept that everything in the system can be represented as a file.

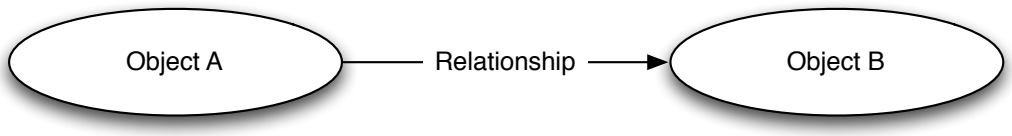

Figure 4.1: Relationship between objects

Some relationships are transitive while others aren't. Permission relationships for security reasons should only be read from left to right and only when the creator of the relationship is the owner of the file or is delegated to be able to do so.

Relationships might not be apparent in many systems. A UNIX file or directory is related to it's owner, it's group and also to everybody. While users and groups in these systems have no semantic meaning (they are simple integers, it is the operating system which assigns meaning to the user ID's). Ordinarily we perhaps don't think about these as relationships but we think about them instead as attributes of the file (which is itself a true statement). Extending the UNIX concept, an ACL list in NTFS is also itself a named relationship. An entity is related to an object by what actions the entity may (or cannot) complete on the object. In this case the relationship is entity—action—object as depicted in Figure 4.2.

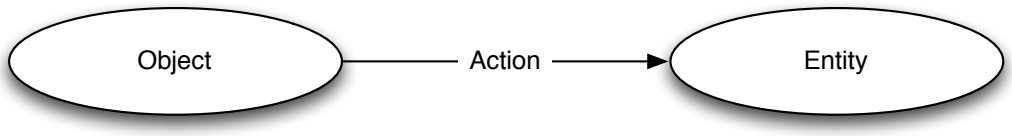

Figure 4.2: An action relationship between an entity and an object

4.2 Current Scenarios

This section lists a set of scenarios modelled off current examples of systems.

4.2.1 UNIX File system Permissions

Permissions in UNIX file systems are familiar to many as it has existed for a significant amount of time. Permissions can be applied to either files or directories. UNIX also extends the concept of 'files' to special files such as character or block devices, pipes and sockets which in terms of access control behave like files. Both 'files' and 'directories' are objects as per 4.1.3.

UNIX object permissions are composed of the following items:

- An owner user and their permissions

- An owner group and their permissions
- Everyone and their permissions

Permissions in UNIX are themselves bits to control read, write and execute. Each individual file stores a copy of the owner's unique user identifier and the unique group identifier which is effectively just a number. This number itself doesn't mean anything to the file system and could identify different users on different systems. The initial implementation of the MDFS system, introduced in Part III, used this technique of storing a simple numeric identifier to identify the owner and group owner of the file.

Building permissions in a metadata system permits us to incorporate users into the file system as a new class type. Users can then have their metadata stored within the file system and the file system can then create relationships between objects in the file system and this new user object. The same concept also applies to groups as well.

So in modelling the UNIX file permissions the following relationships become apparent:

- User owns File
- Group group_owns File

The "everyone" permission can be perhaps defined as an object that is not related to the file by either owns or group_owns, including no relationship at all. Using this we can then apply permissions based on the relationships "owns" and "group_owns". This could be done by using a format which describes the Entity, Action and Object. For example, to emulate the common mode string "rw-r-r–"

- Owner read File
- Owner write File
- Group_Owner read File
- Everyone read File

This has been trivialised however "Owner" in a relational system could be a query to look up the user instead of utilising an attribute as UNIX does. A more detailed examination of the behaviour of ext2, a POSIX compatible file system, is available in 5.1.1.

4.2.2 LDAP Permissions

Lightweight Directory Access Protocol (LDAP) is a protocol that is commonly used to communicate between user directories. LDAP in itself doesn't implement any inherent ACL but the various implementations (e.g. Active Directory ()) handle their own access control. LDAP defines a Distinguished Name (DN) for every object which can be used to reference items in the system. For users the DN also doubles as their username. This is counter intuitive and different systems have different behaviour. Active Directory for example permits a user to login with their full DN, their User Principal Name (UPN) or their Down-level Name () via their LDAP interface.

Many OpenLDAP instances share the common privilege statement listed in Figure 4.2.2 where a user can write or update their own password ("by self write") and everyone can compare the attribute but not necessarily read it ("by * compare"). The example also demonstrates a rule where the user can edit themselves. The

```
access to attr="userpassword"
          by self write
          by * compare
access to * by self write
```

Figure 4.3: Common OpenLDAP ACL

OpenLDAP example demonstrates wildcard matching, the ability to control individual attributes and also the concept of a 'self' identity. As users are stored in the directory as objects it makes sense that the concept of self would exist for the logged in user. In the example provided a wildcard value or "self" is used to identify the entity, the action is either "write" or "compare" and object is either itself or the item attribute "userpassword".

Novell eDirectory is another system that implements LDAP. eDirectory's permissions included the permission "security equivalent to me" and its transitive "security equivalent to X" which flagged that a particular users permissions are equivalent to the target object. These are presented via attributes of the respective users and meant that if the permissions when updated on a user, any user whose security was equivalent to them was also updated. The relevant DN of the user is stored in the attributes. For this the users are both the entity and the object while "security equivalence" is in part the action.

The last aspect is that of group membership. Group membership is typically stored in two ways. In Active Directory, first a user is a "memberOf" a given group and second the group lists each individual "member". Other systems use other attributes (e.g. groupMember and groupMembership attributes in Novell eDirectory stored in the group and user respectively) reinforcing the point that the interpretation of this is not entirely standard. As with the security equivalence, users form both the entity and object.

LDAP is closer to a relational system than perhaps UNIX where all entities in the system (users and groups) are also objects. Relationships are still stored within LDAP as attributes however they are values that are resolvable within the system without relying on an external resource.

4.2.3 Facebook

Facebook has become one of the most prominent social networking sites available today. Facebook itself is a platform for sharing data in new and interesting ways with built in methods of sharing pictures, status updates and events. Facebook provides a third party API that allows people to write applications that interact with Facebook that can store their own data and access the data stored within Facebook.

The ability for third party applications to access information about a user and other issues relating to Facebook's privacy controls have been problematic. Facebook's limited privacy controls are the main point of contention as there is no way to limit exposure of certain items down to certain users. The EFF [81] has noted issues about the diminishing privacy on Facebook and Australian open source developer Paul Fenwick [66] has also documented changes in how Facebook views privacy and the changes it has made.

Facebook has the "self write" privilege granted to permit users to edit stuff about them. Other default permissions appear to be:

- The ability to comment on any item that you can see
- The ability to tag pictures or videos and link these to people you know

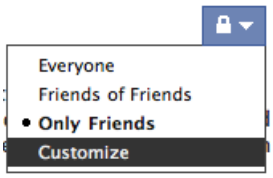

Figure 4.4: Screenshot of Facebook's default security

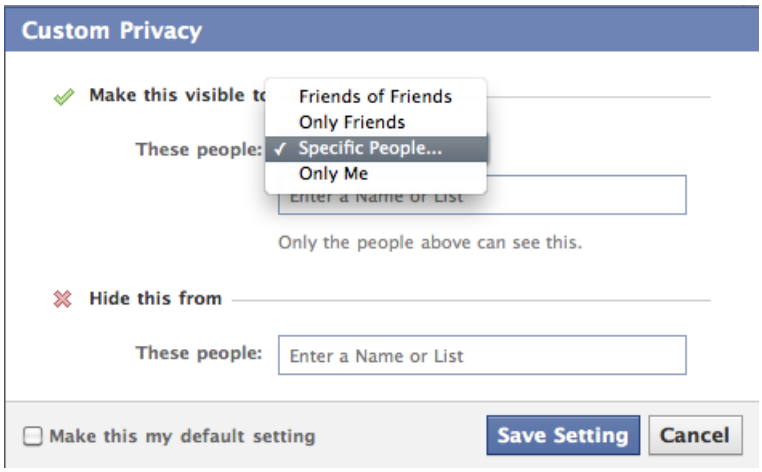

Figure 4.5: Screenshot of Facebook's post security customisation screen

- The ability to untag yourself from pictures or videos

While Facebook has various places to enter information and has various privacy controls, the most obvious controls are when new content is added. This is where Facebook's most powerful access control lists are possible.

Facebook defines the following default groups of posting content:

- Everyone — This is a group which permits full public viewing to all users around the world and on Facebook.
- Only Friends — Only direct friends can see the item being posted.
- Friends of friends — These are your second level contacts. This permits reasonable distribution but limits the level.

The final option, as displayed in Figure 4.4, is to customize the security options. This permits the user to select from the previously available options of Friends of friends and Only Friends with the new option of "Only me" (self) and "Specific People" options. The "specific people" option permits the user to nominate users or lists (lists are analogous to groups, they're a set of users assigned an identifier however lists are per user not global). The screen depicted in Figure 4.5 also provides the ability to specific block users and lists from seeing an item being posted.

Facebook's primary goal for per item access control appears to be providing simple tools that achieve most situations. The Facebook data access control model isn't applied uniformly to all data which in itself presents other issues. The permissions that are there appear increasingly to default to full access to everyone however as demonstrated some parts of the system can be locked down though most of the "static" information is increasingly becoming harder and harder to protect [81].

Facebook has a much more generic access control policy however this appears to be more flexible. As noted, Facebook's security model only limits wall posts and photos [73].

Chapter 5

Examination of Present Systems

5.1 Investigation into the behaviour of file systems

In this section we examine and document the behaviour of two major file systems. The first is the standard ext2/3 file system that ships with Linux operating systems and could be considered an equivalent to the UNIX permissions model. The second is an examination of the NTFS file system. The aim of this is to examine how each file system behaved when evaluating permissions as UNIX doesn't have the concept of inheritable permissions (all permissions are explicitly granted) as opposed to NTFS which does feature full inheritable permissions. UNIX does however support the ability for owner and group inheritance for a file in a particular folder but this does not apply to any subfolder unless explicitly set.

5.1.1 ext2/3 behaviour

To begin our experimental we will create a directory structure which mimics a collection of pie recipes:

```
817621 drwxr-xr-x 4 pasamio pasamio pies/
817796 drwxr-xr-x 2 pasamio pasamio pies/savoury/
817798 -rw-r--r-- 1 pasamio pasamio pies/savoury/beef
817799 -rw-r--r-- 1 pasamio pasamio pies/savoury/chicken
817800 -rw-r--r-- 1 pasamio pasamio pies/savoury/floater
817801 -rw-r--r-- 1 pasamio pasamio pies/savoury/fish
817802 -rw-r--r-- 1 pasamio pasamio pies/savoury/pork
817803 drwxr-xr-x 2 pasamio pasamio pies/sweet/
817804 -rw-r--r-- 1 pasamio pasamio pies/sweet/apple
817805 -rw-r--r-- 1 pasamio pasamio pies/sweet/blackberry
817806 -rw-r--r-- 1 pasamio pasamio pies/sweet/custard
817807 -rw-r--r-- 1 pasamio pasamio pies/sweet/pumpkin
817809 -rw-r--r-- 1 pasamio pasamio pies/sweet/cream
```

* Note: This is the output of ls -idlF with the size, date and time fields removed.

As we can see the files have been created with the default permissions for my user account and is owned by my account and my group (pasamio for both). By default all users can read the files or directories and can also traverse the directories. At this point we can list any directory, traverse it and edit any of the files listed.

If we remove the "read" permission from "sweet" directory and attempt to list the contents of sweet we are met with a "Permission denied" error:

```
pasamio@fallenwall:/tmp/pies$ chmod -r sweet
pasamio@fallenwall:/tmp/pies$ ls sweet
ls: sweet: Permission denied
```

However, if we know the name of a file then we can open it, write to it and read it:

```
pasamio@fallenwall:/tmp/pies$ echo Apple Pie > sweet/apple
pasamio@fallenwall:/tmp/pies$ cat sweet/apple
Apple Pie
```

If we remove the directory traversal (execute) bit for that directory however, we can't do anything:

```
pasamio@fallenwall:/tmp/pies$ chmod -x sweet/
pasamio@fallenwall:/tmp/pies$ ls sweet
ls: sweet: Permission denied
pasamio@fallenwall:/tmp/pies$ cat sweet/apple
cat: sweet/apple: Permission denied
```

If at this point we store read access to the directory, we regain the ability to list the files in it but can't do anything with that information as we don't get inodes:

```
pasamio@fallenwall:/tmp/pies$ chmod +r sweet
pasamio@fallenwall:/tmp/pies$ ls sweet
apple  blackberry  cream  custard  pumpkin
pasamio@fallenwall:/tmp/pies$ ls -i sweet
0 sweet/apple
0 sweet/blackberry
0 sweet/cream
0 sweet/custard
0 sweet/pumpkin
pasamio@fallenwall:/tmp/pies$ cat sweet/apple
cat: sweet/apple: Permission denied
```

At this point I would like to note that while read and traversal have been removed, write has not. However I cannot create a new entry unless traversal has been restored:

```
pasamio@fallenwall:/tmp/pies$ echo Pecan > sweet/pecan
-bash: sweet/pecan: Permission denied
pasamio@fallenwall:/tmp/pies$ chmod +X sweet/
pasamio@fallenwall:/tmp/pies$ echo Pecan > sweet/pecan
pasamio@fallenwall:/tmp/pies$ cat sweet/pecan
Pecan
```

If at this point write access is removed, the ability to create new entries is also removed:

```
pasamio@fallenwall:/tmp/pies$ chmod -w sweet
pasamio@fallenwall:/tmp/pies$ echo Buttermilk > sweet/buttermilk
-bash: sweet/buttermilk: Permission denied
```

If we restore permissions to their original values we can also examine the way links behave with regards to traversal. Ordinarily we can traverse each directory with ease and this is controlled by the execute bit or the directory traversal bit. To prove this we can remove the read permission to a directory but still change to it:

```
pasamio@fallenwall:/tmp/pies$ chmod -r sweet
pasamio@fallenwall:/tmp/pies$ cd sweet
pasamio@fallenwall:/tmp/pies/sweet$ cd ..
pasamio@fallenwall:/tmp/pies$
```

However if we restore the read bit and remove the traversal bit I cannot change to it:

```
pasamio@fallenwall:/tmp/pies$ chmod +r sweet
pasamio@fallenwall:/tmp/pies$ chmod -x sweet
pasamio@fallenwall:/tmp/pies$ cd sweet
-bash: cd: sweet: Permission denied
```

This applies to any child of that directory also. So if we restore traversal to "sweet" and remove it from "pies", we cannot switch to the sweet directory any more:

```
 pasamio@fallenwall:/tmp$ chmod -x pies
pasamio@fallenwall:/tmp$ cd pies/sweet
-bash: cd: pies/sweet: Permission denied
```

Now to prove that permissions work through tree traversal, we will create a new folder at the same level as "pies" called "recipes" and create symbolic links to the files:

```
pasamio@fallenwall:/tmp$ mkdir recipes
pasamio@fallenwall:/tmp$ cd recipes/
pasamio@fallenwall:/tmp/recipes$ ls
pasamio@fallenwall:/tmp/recipes$ ln -s ../pies/sweet/ sweet\ pies
pasamio@fallenwall:/tmp/recipes$ ln -s ../pies/savoury/chicken chicken\ pie
pasamio@fallenwall:/tmp/recipes$ ls
chicken pie  sweet pies
```

Now if the traversal bit is removed from the "pies" directory, we can examine the impact:

```
pasamio@fallenwall:/tmp/recipes$ cd ../
pasamio@fallenwall:/tmp$ chmod -x pies
pasamio@fallenwall:/tmp$ ls recipes/sweet\ pies
ls: recipes/sweet pies: Permission denied
pasamio@fallenwall:/tmp$ ls recipes/chicken\ pie
ls: recipes/chicken pie: Permission denied
```

Since symbolic links were used, the path is traversed down the link to obtain the file. As "pies" does not permit traversal this results in an error. Restoring the permissions we can this time create a hard link to the recipe file again and then remove (also note that hard links for directories aren't permitted using Linux [25] but are permitted as an optional implementation feature in POSIX [33]; Mac OS X supports directory hard links as of Mac OS X 10.5, see 5.1.2):

```
pasamio@fallenwall:/tmp$ chmod +x pies
pasamio@fallenwall:/tmp$ rm -rf recipes/chicken\ pie
pasamio@fallenwall:/tmp$ rm -rf recipes/sweet\ pies
pasamio@fallenwall:/tmp$ ln pies/sweet/chicken recipes/chicken\ pie
pasamio@fallenwall:/tmp$ ln pies/sweet/ recipes/sweet\ pies
ln: ‘pies/sweet/’: hard link not allowed for directory
```

So if we remove directory traversal (execute bit for directories) on "pies" again, we can examine the result:

```
pasamio@fallenwall:/tmp$ chmod -x pies
pasamio@fallenwall:/tmp$ cat recipes/chicken\ pie
Chicken
```

So when we hard link to the file, we use the directory path to get to that given hard link to evaluate its permissions. This doesn't appear to be a major issue because the Linux system call "link" only supports file names.

5.1.2 HFS+

HFS+ is the file system used by Apple for their Mac OS X operating system. HFS+ has very similar features to that offered by ext2 with HFS+ supporting hard linked directories [68]. With this in mind, consider the example "pies" directory and "recipes" with a hard link created to the "sweet" subdirectory of "pies" (hlink is a third party application [74]):

```
silversaviour:tmp pasamio$ hlink pies/sweet/ recipes/sweet\ pies
silversaviour:tmp pasamio$ chmod -x pies
silversaviour:tmp pasamio$ cd recipes/sweet\ pies/
silversaviour:sweet pies pasamio$ ls
apple blackberry cream custard pumpkin
```

As with the files under Linux, the user is able to get to a location based on having a hard link to the location while another pathway is blocked. Removing traversal from the "sweet" directory denies access from any location as the directory maintains its permissions data itself:

```
silversaviour:tmp pasamio$ chmod +x pies
silversaviour:tmp pasamio$ chmod -x pies/sweet
silversaviour:tmp pasamio$ cd recipes/sweet\ pies/
-bash: cd: recipes/sweet pies/: Permission denied
```

As we can see from this, access control is determined by the traversal path and if an alternative traversal path is used then access will be granted.

It should be noted that the Apple documentation for the link system call [58] states that "As mandated by POSIX.1, path1 may not be a directory." As of Mac OS X 10.5, if this is executed on a HFS+ file system then a directory name may be used as is demonstrated above. If this is executed on a non-HFS+ file system, the system call errors with the result being "Operation not permitted" instead of the link being created.

5.1.3 NTFS

For NTFS, a Windows XP Professional installation was used with a similar set of tests as per UNIX. Since NTFS supports inheritance, the key investigative point is to examine how this works with explicitly set permissions at different levels of the tree. As with UNIX, the "pies" sample directory layout is used again. Since Windows XP Professional doesn't appear to ship with a tool to examine ACL permissions by default, the "FILEACL" tool has been used, accessible from Microsoft's web site [18]. The machine is named "WINXP" and the account used is "FAIL Administrator" or just "Administrator".

In experimenting with Windows, the tests that were taken out under UNIX will not be undertaken. With the sample directory that has been created, we can examine its permissions:

```
C:\research\pies>fileacl sweet
C:\research\pies\sweet;BUILTIN\Users:WwA/U[I]
C:\research\pies\sweet;BUILTIN\Administrators:F[I]
C:\research\pies\sweet;NT AUTHORITY\SYSTEM:F[I]
C:\research\pies\sweet;WINXP\Administrator:F/U/U[I]
C:\research\pies\sweet;CREATOR OWNER:U/F/F[I]
C:\research\pies\sweet;BUILTIN\Users:RX[I]
```

So using fileacl's help output references [4], this can be deciphered into something more readable.

- Users have a unset permission on files (U), can add files to the directory (Ww) add a subdirectory (A) for this folder and subfolders. Users also have another permission which grants them the normal ability to read the directory or files, read attributes, read extended attributes and read permissions (R) as well as traverse the directory or execute the file (X) for this folder, subfolders and child files.
- Administrators (group), Administrator (user), the SYSTEM account, CREATOR OWNER have full control (F). However only Administrators and SYSTEM has it for "this folder, subfolders and files". Administrator only has this right for this folder and CREATOR OWNER has this right for "subfolders and files only".
- All permissions are "autopropagated" or inherited ([I]) from a parent entry.

Setting the deny bit for "full control" to the "pies" directory results in the following ouptut:

```
C:\research>fileacl pies
C:\research\pies;DENY!WINXP\Administrator:RWwAWaWePXDDcO
C:\research\pies;BUILTIN\Administrators:F[I]
C:\research\pies;NT AUTHORITY\SYSTEM:F[I]
C:\research\pies;WINXP\Administrator:F/U/U[I]
C:\research\pies;CREATOR OWNER:U/F/F[I]
C:\research\pies;BUILTIN\Users:RWwAX/RX[I]
```

Most other groups appear unaffected, for some reason the Users group appears to have modified its permissions slightly at this level however it is apparent that the Administrator now is denied from almost every permission that is available to the entry and any child entries. The lack of a [I] denotes that this is an explicit permission set at this level and isn't inherited. Attempts to access children or change to the directory aren't permitted:

```
C:\research>cd pies
Access is denied.

C:\research>more pies\sweet\apple
Cannot access file C:\research\pies\sweet\apple
```

Reverting those assigned permissions back, apply now an explicit grant of full control to the "sweet" directory and then an explicit deny to the "pies" directory:

```
C:\research>fileacl pies
C:\research\pies;DENY!WINXP\Administrator:RWwAWaWePXDDcO
C:\research\pies;BUILTIN\Administrators:F[I]
C:\research\pies;NT AUTHORITY\SYSTEM:F[I]
```

```
C:\research\pies;WINXP\Administrator:F/U/U[I]
C:\research\pies;CREATOR OWNER:U/F/F[I]
C:\research\pies;BUILTIN\Users:RWwAX/RX[I]

C:\research>fileacl pies\sweet
C:\research\pies\sweet;DENY!WINXP\Administrator:RWwAWaWePXDDcO[I]
C:\research\pies\sweet;WINXP\Administrator:F
C:\research\pies\sweet;BUILTIN\Administrators:F[I]
C:\research\pies\sweet;NT AUTHORITY\SYSTEM:F[I]
C:\research\pies\sweet;WINXP\Administrator:F/U/U[I]
C:\research\pies\sweet;CREATOR OWNER:U/F/F[I]
C:\research\pies\sweet;BUILTIN\Users:RWwAX/RX[I]

C:\research>more pies\sweet\apple
Apple

C:\research>cd pies
Access is denied.

C:\research>fileacl pies\savoury
GetFileAtt error! (rc=5)
Access is denied.
can't get dir attribute, try /FORCE
C:\research\pies\savoury;DENY!WINXP\Administrator:RWwAWaWePXDDcO[I]
C:\research\pies\savoury;BUILTIN\Administrators:F[I]
C:\research\pies\savoury;NT AUTHORITY\SYSTEM:F[I]
C:\research\pies\savoury;WINXP\Administrator:F/U/U[I]
C:\research\pies\savoury;CREATOR OWNER:U/F/F[I]
C:\research\pies\savoury;BUILTIN\Users:RWwAX/RX[I]

C:\research>more pies\savoury\beef
Cannot access file C:\research\pies\savoury\beef
```

Examining this there are a few things happening:

- The "pies" directory again has the same permissions that prevent access to the sweet directory - so much so that the contents of the "pies" directory cannot be listed nor can it be traversed.
- The "sweet" directory notes that Administrator has a DENY ACL entry against it that is inherited however it has an explicit full control assigned to that level.
- The "savoury" directory initially returns a Access is denied error before fileacl uses an force override to retrieve the permissions via a backup privilege.
- Members of the "sweet" directory are accessible however members of the "savoury" directory are not.

Thus it can be noted that NTFS appears to actually traverse the tree backwards from the object since the "pies" directory would have denied this from occurring under Linux without the existence of a hard link. This behaviour is because of the method that NTFS uses to evaluate file permissions using the DACL system where each layer is inspected before progressing up the chain.

Curiously Microsoft presents a warning when assigning deny permissions which might mislead an administrator that doesn't realise that the deny permission they are setting could potentially be overruled by an allow entry at a lower level.

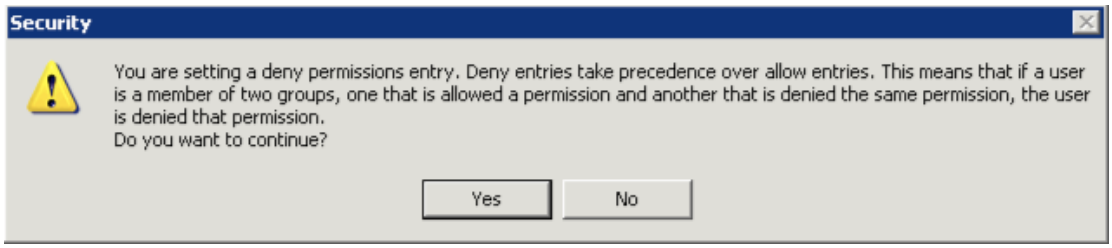

Figure 5.1: Warning about deny privilege from Microsoft Windows XP

NTFS supports hard linking [8] for POSIX compliance [17]. Just like Linux however, hard links for directories don't appear to be possible however hard links for files are possible:

```
C:\research\recipes>fsutil hardlink create "c:\research\recipes\sweet pies"
        c:\research\pies\sweet
Error:  Access is denied.

C:\research\recipes>fsutil hardlink create "c:\research\recipes\blackberry pie"
        c:\research\pies\sweet\blackberry
Hardlink created for
        c:\research\recipes\blackberry pie <<===>> c:\research\pies\sweet\blackberry

C:\research\recipes>more "blackberry pie"
Blackberry
```

NOTE: Output artificially altered to fit page width

So with the files are linked to each other, the permission to access the "pies" directory is revoked:

```
C:\research>fileacl pies
C:\research\pies;DENY!WINXP\Administrator:RWwAWaWePXDDcO
C:\research\pies;BUILTIN\Administrators:F[I]
C:\research\pies;NT AUTHORITY\SYSTEM:F[I]
C:\research\pies;WINXP\Administrator:F/U/U[I]
C:\research\pies;CREATOR OWNER:U/F/F[I]
C:\research\pies;BUILTIN\Users:RWwAX/RX[I]

C:\research>cd pies
Access is denied.

C:\research>more "recipes\blackberry pie"
Cannot access file C:\research\recipes\blackberry pie
```

In this case, unlike with Linux, permission to access the file has been disabled just as if it had been accessed via the normal means. Since the DACL is constructed in the object appears to only support a single line of inheritance.

A final curious feature is the way that inherited permissions are handled. To open a file in a directory you need rights over that file to read it but you also need the "List Files/Read Data" permission from the containing folder. However to open a folder, this permission isn't required. So if you have list files on the target folder you can read it regardless of permissions from parents and without requiring directory traversal rights. Consider the following configuration:

```
C:\research>fileacl pies
C:\research\pies;DENY!FAIL\Administrator:RWwAWaWePXDDcO
C:\research\pies;BUILTIN\Administrators:F[I]
C:\research\pies;NT AUTHORITY\SYSTEM:F[I]
C:\research\pies;FAIL\Administrator:F/U/U[I]
C:\research\pies;CREATOR OWNER:U/F/F[I]
C:\research\pies;BUILTIN\Users:RWwAX/RX[I]

C:\research>fileacl pies\sweet /FORCE
C:\research\pies\sweet;DENY!FAIL\Administrator:RaRepWwAWaWePXDDcO
C:\research\pies\sweet;FAIL\Administrator:Rr

C:\research>fileacl pies\sweet\apple
C:\research\pies\sweet\apple;FAIL\Administrator:F
C:\research\pies\sweet\apple;BUILTIN\Administrators:F
C:\research\pies\sweet\apple;NT AUTHORITY\SYSTEM:F
C:\research\pies\sweet\apple;BUILTIN\Users:RX
```

So the "pies" directory has a deny set for the Administrator account. The "pies\sweet" directory has a deny for almost everything except the "list files/read data" permission (Rr). Finally the file "pies\sweet\apple" has full control granted. With these rights we are able to read the file:

```
C:\research>more pies\sweet\apple
Apple
```

However if the Rr permission is removed from "pies
sweet", we cannot access the file any more:

```
C:\research>fileacl pies\sweet /FORCE
C:\research\pies\sweet;DENY!FAIL\Administrator:RaRepWwAWaWePXDDcO

C:\research>more pies\sweet\apple
Cannot access file C:\research\pies\sweet\apple
```

This appears to be the result of the SeChangeNotifyPrivilege privilege in action. The SeChangeNotifyPrivilege, detailed in 3.4.5, provides the ability to skip directory traversal checks. So even though the parent would ordinarily deny access as this isn't being checked as a matter of performance, access is permitted to the directory. It is curious to note that this privilege is granted to all users in Windows. Without this privilege the behaviour seen within Unix, Linux and Mac OS X (e.g. of being denied access to a child if traversal is removed from a parent at any point in the chain) would be implemented. It would appear that removing this privilege can potentially cause many issues with a Windows server, particularly with file sharing [86].

5.2 Implementation of NTFS by means of XACML

5.2.1 Sample of NTFS Access Rules

Consider the following set of rules:

- User john is explicitly unable to read the file "/Reports/Annual Report.txt"
- Group "Report Authors" is able to read and write to "/Reports/" which is inherited to all children

- User fred has an explicit deny to write to "/Reports/" which isn't inherited.
- User fred can read the file "/Reports/Annual Report.txt"
- Group "Report Authors" has the users john and harry

For simplicity we will only deal with the generic rights. A rules for NTFS are expressed as highly restrictive data structures, I'm going to use the description syntax used by fileacl [4] .

```
/Reports/Annual Report.txt;DENY!john:R
/Reports:Report Authors:RW/RW/RW
/Reports;fred:DENY!W/U/U
/Reports/Annual Report.txt;fred:R
```

Note: As XACML only designates the use of URI or UNIX fie system paths, this style has been used instead of the traditional MS-DOX drive letter style syntax.

5.2.2 Sample of NTFS Access Rules in XACML

Two policy sets need to be defined for "/Reports" - one that is inherited and another which isn't. From here there needs to be a policy for "/Reports/Annual Report.txt" which defines the rules for this file and then also calls the inherited values from "/Reports". As the only way to reference other sets of rules is via a PolicySetIdReference or a PolicyIdReference, either of which having to be in a PolicySet themselves. So in this case the approach of using a PolicySetIdReference was used to reference an entire PolicySet which could in turn reference other policies. This way NTFS style reverse traversal of the tree from the child to the root can be achieved.

Policy Set 1: Inheritable /Reports permissions

```
<?xml version="1.0" encoding="UTF-8" ?>
<PolicySet
  xmlns="urn:oasis:names:tc:xacml:1.0:policy"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  PolicySetId="urn:au:edu:usq:example:ntfs:policysetid:1"
  PolicyCombiningAlgId=
  "urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:first-applicable" />
  <Description>
    Policy Set for NTFS Example (/Reports)
  </Description>
  <Target>
    <Subjects>
      <Subject>
        <AnySubject />
      </Subject>
    </Subjects>
    <Resources>
      <AnyResource />
    </Resources>
    <Actions>
      <AnyAction />
    </Actions>
  </Target>
  <PolicySetIdReference>
    urn:au:edu:usq:example:ntfs:policysetid:owner
```

```
</PolicySetIdReference>
<Policy
  PolicyId="urn:au:edu:usq:example:ntfs:policyid:1"
  RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:first-applicable">
  <Description>
    Inheritable Policy for /Reports
  </Description>
  <Target>
    <Subjects>
      <Subject>
        <AnySubject />
      </Subject>
    </Subjects>
    <Resources>
      <AnyResource />
    </Resources>
    <Actions>
      <AnyAction />
    </Actions>
  </Target>
  <Rule
    RuleId="urn:au:edu:usq:example:ntfs:Rule1"
    Effect="Permit">
    <Description>
      Inheritable Policy for /Reports
    </Description>
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <SubjectAttributeDesignator
              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
              DataType="http://www.w3.org/2001/XMLSchema#string" />
                <AttributeValue
                  DataType="http://www.w3.org/2001/XMLSchema#string">
                    Report Authors
                </AttributeValue>
            </SubjectAttributeDesignator>
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources>
        <Resource>
          <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:ufs-path"
            DataType="http://www.w3.org/2001/XMLSchema#anyURI">
            <AttributeValue>
              /Reports
            </AttributeValue>
          </Attribute>
          <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:scope"
            DataType="http://www.w3.org/2001/XMLSchema#string">
            <AttributeValue>
              Descendants
            </AttributeValue>
```

```
            </Attribute>
          </Resource>
        </Resources>
        <Actions>
          <Action>
            <ActionMatch
              MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
                <AttributeValue
                  DataType="http://www.w3.org/2001/XMLSchema#string">
                  read
                </AttributeValue>
              <ActionAttributeDesignator AttributeId=
                "urn:oasis:names:tc:xacml:1.0:action:action-id"
                DataType="http://www.w3.org/2001/XMLSchema#string" />
            </ActionMatch>
            <ActionMatch
              MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
                <AttributeValue
                  DataType="http://www.w3.org/2001/XMLSchema#string">
                  write
                </AttributeValue>
              <ActionAttributeDesignator AttributeId=
                "urn:oasis:names:tc:xacml:1.0:action:action-id"
                DataType="http://www.w3.org/2001/XMLSchema#string" />
            </ActionMatch>
          </Action>
        </Actions>
      </Target>
    </Rule>
  </Policy>
</PolicySet>
```

Policy Set 2: /Reports/Annual Report.txt

```
<?xml version="1.0" encoding="UTF-8" ?>
<PolicySet
  xmlns="urn:oasis:names:tc:xacml:1.0:policy"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  PolicySetId="urn:au:edu:usq:example:ntfs:policysetid:2"
  PolicyCombiningAlgId=
  "urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:first-applicable" />
  <Description>
    Policy Set for NTFS Example (/Reports/Annual Report.txt)
  </Description>
  <Target>
    <Subjects>
      <Subject>
        <AnySubject />
      </Subject>
    </Subjects>
    <Resources>
      <AnyResource />
    </Resources>
    <Actions>
      <AnyAction />
```

```
    </Actions>
  </Target>
  <PolicySetIdReference>
    urn:au:edu:usq:example:ntfs:policysetid:owner
  </PolicySetIdReference>
  <Policy
    PolicyId="urn:au:edu:usq:example:ntfs:policyid:2"
    RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:first-applicable">
    <Description>
      Policy for /Reports/Annual Report.txt
    </Description>
    <Target>
      <Subjects>
        <Subject>
          <AnySubject />
        </Subject>
      </Subjects>
      <Resources>
        <AnyResource />
      </Resources>
      <Actions>
        <AnyAction />
      </Actions>
    </Target>
    <Rule
      RuleId="urn:au:edu:usq:example:ntfs:Rule2"
      Effect="Deny">
      <Description>
        User john is explicitly unable to read the file /Reports/Annual Report.txt
      </Description>
      <Target>
        <Subjects>
          <Subject>
            <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <SubjectAttributeDesignator
                AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
                DataType="http://www.w3.org/2001/XMLSchema#string" />
                  <AttributeValue
                    DataType="http://www.w3.org/2001/XMLSchema#string">
                      john
                  </AttributeValue>
              </SubjectAttributeDesignator>
            </SubjectMatch>
          </Subject>
        </Subjects>
        <Resources>
          <Resource>
            <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:ufs-path"
              DataType="http://www.w3.org/2001/XMLSchema#anyURI">
              <AttributeValue>
                /Reports/Annual Report.txt
              </AttributeValue>
            </Attribute>
          </Resource>
```

```
      </Resources>
      <Actions>
        <Action>
          <ActionMatch
            MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue
                DataType="http://www.w3.org/2001/XMLSchema#string">
                read
              </AttributeValue>
            <ActionAttributeDesignator AttributeId=
              "urn:oasis:names:tc:xacml:1.0:action:action-id"
              DataType="http://www.w3.org/2001/XMLSchema#string" />
          </ActionMatch>
        </Action>
      </Actions>
    </Target>
  </Rule>
  <Rule
    RuleId="urn:au:edu:usq:example:ntfs:Rule3"
    Effect="Permit">
    <Description>
      User fred can read the file /Reports/Annual Report.txt
    </Description>
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <SubjectAttributeDesignator
              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
              DataType="http://www.w3.org/2001/XMLSchema#string" />
                <AttributeValue
                  DataType="http://www.w3.org/2001/XMLSchema#string">
                    fred
                </AttributeValue>
            </SubjectAttributeDesignator>
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources>
        <Resource>
          <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:ufs-path"
            DataType="http://www.w3.org/2001/XMLSchema#anyURI">
            <AttributeValue>
              /Reports/Annual Report.txt
            </AttributeValue>
          </Attribute>
        </Resource>
      </Resources>
      <Actions>
        <Action>
          <ActionMatch
            MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue
                DataType="http://www.w3.org/2001/XMLSchema#string">
```

```
                read
              </AttributeValue>
            <ActionAttributeDesignator AttributeId=
              "urn:oasis:names:tc:xacml:1.0:action:action-id"
              DataType="http://www.w3.org/2001/XMLSchema#string" />
          </ActionMatch>
        </Action>
      </Actions>
    </Target>
  </Rule>
</Policy>
<!-- Reference to /Reports -->
<PolicySetIdReference>
  urn:au:edu:usq:example:ntfs:policysetid:1
</PolicySetIdReference>
</PolicySet>
```

Policy Set 3: /Reports non-inherited permissions

```
<?xml version="1.0" encoding="UTF-8" ?>
<PolicySet
  xmlns="urn:oasis:names:tc:xacml:1.0:policy"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  PolicySetId="urn:au:edu:usq:example:ntfs:policysetid:3"
  PolicyCombiningAlgId=
  "urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:first-applicable" />
  <Description>
    Policy Set for NTFS Example (/Reports/Annual Report.txt)
  </Description>
  <Target>
    <Subjects>
      <Subject>
        <AnySubject />
      </Subject>
    </Subjects>
    <Resources>
      <AnyResource />
    </Resources>
    <Actions>
      <AnyAction />
    </Actions>
  </Target>
  <PolicySetIdReference>
    urn:au:edu:usq:example:ntfs:policysetid:owner
  </PolicySetIdReference>
  <Policy
    PolicyId="urn:au:edu:usq:example:ntfs:policyid:3"
    RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:first-applicable">
    <Description>
      Policy for /Reports/Annual Report.txt
    </Description>
    <Target>
      <Subjects>
        <Subject>
          <AnySubject />
```

```
      </Subject>
    </Subjects>
    <Resources>
      <AnyResource />
    </Resources>
    <Actions>
      <AnyAction />
    </Actions>
  </Target>
  <Rule
    RuleId="urn:au:edu:usq:example:ntfs:Rule4"
    Effect="Deny">
    <Description>
      User fred can't write to /Reports but this isn't inherited
    </Description>
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <SubjectAttributeDesignator
              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
              DataType="http://www.w3.org/2001/XMLSchema#string" />
                <AttributeValue
                  DataType="http://www.w3.org/2001/XMLSchema#string">
                    fred
                </AttributeValue>
            </SubjectAttributeDesignator>
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources>
        <Resource>
          <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:ufs-path"
            DataType="http://www.w3.org/2001/XMLSchema#anyURI">
            <AttributeValue>
              /Reports/Annual Report.txt
            </AttributeValue>
          </Attribute>
          <!-- Immediate scope is implied by not setting a scope
              but this makes it explicit -->
          <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:scope"
            DataType="http://www.w3.org/2001/XMLSchema#string">
            <AttributeValue>
              Immediate
            </AttributeValue>
          </Attribute>
        </Resource>
      </Resources>
      <Actions>
        <Action>
          <ActionMatch
            MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue
                DataType="http://www.w3.org/2001/XMLSchema#string">
```

```
                write
              </AttributeValue>
            <ActionAttributeDesignator AttributeId=
              "urn:oasis:names:tc:xacml:1.0:action:action-id"
              DataType="http://www.w3.org/2001/XMLSchema#string" />
          </ActionMatch>
        </Action>
      </Actions>
    </Target>
  </Rule>
 </Policy>
</PolicySet>
```

5.3 Examination of Privileges and Capabilities

Privileges and capabilities are features offered by the Windows Security Model and the Linux Security Model. They provide for the ability to control access to parts of the system. A key difference is that the Windows privilege model grants users the ability to complete tasks from no ability while the Linux capability system does the same but is aimed at going from full capability to limited capability.

Privileges form a mandatory access control model. Privileges and capabilities will override any discretionary permissions that could exist. A good example of this is the capability within Unix systems that root can bypass most discretionary access controls to read and write files on the local file system. Privileges and capabilities also provide access to items that aren't normally controlled through other discretionary measures. An example of this is in Windows the ability to manage volumes (disks) or in Unix/Linux the ability to bind to privileged ports.

The Windows privilege model has various abilities that can be assigned to users of groups. An example of two such privileges is the ability to backup a file or to restore it. These abilities map directly onto a privilege that can be granted within Windows. No user within Windows inalienably has these privileges and they can be configured to be granted, or not granted, to individual users. Various privileges relating to file systems are noted in the NTFS subsection on privileges (see 3.4.5). Privileges provide abilities to normal users.

The Linux capability model appears to have been partially implemented and not completed fully. The Linux capability isn't as rich as the Windows privilege model. The Linux capability system appears to work by starting with a root user that has all capabilities and then progressively removing capabilities. This means that processes can remove capabilities that they don't require (e.g. remove the capability to ignore file permissions) and retain those capabilities that they do need (e.g. to send raw packets or to bind to a privileged port). However to do this the application either needs itself to utilise the capability system to drop surplus capabilities or it needs to be run through a wrapper which removes the capabilities for the process. Capabilities haven't been used much since their introduction in 1998 and while there have been attempts to complete the work on capabilities it appears to be a contentious area [64].

5.4 Comparisons

Each of the file systems present various features. Some present general options and others don't. This section is split into two categories: generic permission based features of systems and a specific section which covers file system permissions.

	FAT*	NTFS	ext2/3	POSIX ACL	HFS+	NFSv4 ACL	ZFS	NWFS
Basic File Permissions								
Write	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Read	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Execute	No	Yes	Yes	Yes	Yes	Yes	Yes	Attr Only
Read-only	Attr Only	Yes	Attr Only	No	Yes (Locked)	No	No	Yes
Hidden	Attr Only	Both*	No	No	No	Yes	Yes	Both
Basic Directory Permissions								
List	No	Yes*	Yes	Yes	Yes	Yes	Yes	Both
Search	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Add/Delete child	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Attribute Permissions								
Attribute Read	No	Yes	No	No	Yes	Yes	Yes	No
Attribute Write	No	Yes	No	No	Yes	Yes	Yes	No
Extended Attribute Write	No	Yes	No	No	Yes	Yes	Yes	No
Extended Attribute Read	No	Yes	No	No	Yes	Yes	Yes	No
Read Permissions	No	Yes	No	No	Yes	Yes	Yes	No
Write Permissions	No	Yes	No	No	Yes	Yes	Yes	Yes
Change Ownership	No	Yes	No	Yes	Yes	Yes	Yes	No
Extended File Permissions								
Delete	No	No	Attr Only*	No	No	No	No	Attr Only
Append	No	No	Yes	No	No	No	No	No
Extended Directory Permission								
Add file	No	Yes	No	No	Yes	Yes	Yes	Yes
Add directory	No	Yes	No	No	Yes	Yes	Yes	Yes
Add subdirectory	No	Yes	No	No	Yes	Yes	Yes	Yes
Delete child	No	Yes	No	No	Yes	Yes	Yes	Yes
Specialised Permissions								
Supervisor	No	O/S	O/S	O/S	O/S	No	O/S	Yes
Rename Inhibit	No	No	No	No	No	No	No	Attr Only
Copy Inhibit	No	No	No	No	No	No	No	Attr Only
System	Attr Only	Attr Only	No	No	No	No	No	Attr Only
Access Control	No	O/S	No	No	No	No	No	Yes
Stationary	No	No	No	No	Attr Only	No	No	No

Table 5.1: Comparison of functionality between various file systems

Legend:

- Yes — The permission is controllable on a per item basis for this flle system via a particular permission.
- Both — The permission is controllable on a per item basis for this file system using either permissions or attributes.
- Attr Only — The permission is controllable on a per item basis for this operating system via a particular attribute. Attributes apply to all users equally.
- No — The permission cannot be controlled in any way. The permission may also be masked by other permissions however (e.g. extra read permissions like read permissions or read attributes may be a part of a more general read permission) but cannot be controlled individually and are marked as no.
- O/S — The particular permission is not controllable within the file system however the operating system offers the ability to permit equivalent controls either via capabilities or privileges.

Notes:

- FAT is a single user file system, all "permissions" listed are implemented through attributes which thus apply to all users.
- NTFS implements the hidden attribute which hides a particular file from view depending on the users settings and also offers ABE. NetWare File System (NWFS) supports both a hidden attribute and a "file scan" permission that prohibits users seeing a file or folder.
- ext2 has an immutable bit that specifies that a file cannot be altered by anyone, including any privileged superuser which is equivalent to read-only and delete inhibit.
- HFS+ here refers to the version shipped with Mac OS X 10.4 (Tiger) or greater which includes support for POSIX ACLs. Versions previous to this implemented the standard POSIX security model.

Interestingly, the disparate security and permission features implemented by each file system can be structured into a graph of where their primary features are derived from. In some aspects of the graph, a line is drawn when a system logically extends the system, such is the case of NWFS which was derived from FAT. More commonly arrows are drawn when features are similarly developed. This can be by implementing a common standard (e.g. POSIX.1 compliance) or implementing a very similar security model (such is the case of NFSv4 which has a similar model to that found in NTFS) or aspects of the security model (e.g. HPFS's extended attributes were implemented in NTFS for support).

The graph in Figure 5.2 has root nodes as either the operating system that implemented the particular file system (e.g. Netware, OS/2, Windows/DOS) or a particular standard (e.g. POSIX, used by UNIX and its clone operating systems such as Linux).

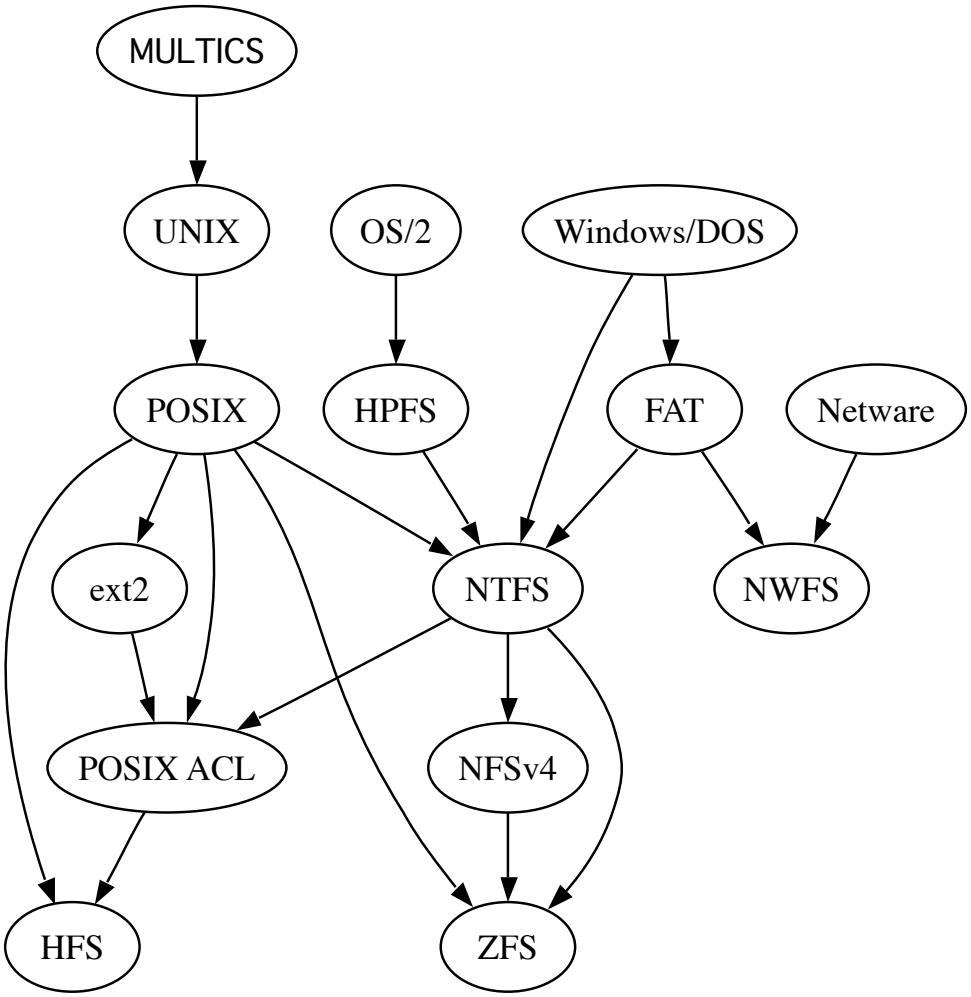

Figure 5.2: Family tree of file systems and access control standards

Part III

Experimentation

Chapter 6

MDFS Purpose

This chapter puts in perspective the purpose of MDFS and examining the utilisation of rich metadata in access control. It covers why metadata is an important part of access control and how building systems with access to rich metadata can provide more scalable access control systems. The chapter continues to provide a background in metadata file systems and how metadata has been treated in existing and historic systems. It concludes by introducing the goals for the development of an MDFS undertaken in this project.

6.1 The importance of metadata in access control

As discussed in Chapter 2, many of the access control systems that exist today can be viewed as metadata that is layered on top of the objects that they are protecting. In some cases this is a very small amount of data (32 bits of mask for UNIX plus the owner and group of an object) and in other cases it a comparatively large amount of data (up to 64KB for NTFS). However all of these systems are constrained by the limitations on the range of information which is accessible.

Many access control systems examined are designed to operate in a way where they don't know semantically what they are protecting. When you set an ACL on a file that says that your relation can see a file, the access control system doesn't understand why you're doing this and nor does it attempt to work out why. This leads to the situation where you are continually granting access to items that people need access to because there is no semantic meaning behind the access control rights you are assigning.

To a point this is mostly alleviated in corporate environments by creating folders on network file shares that have the permissions assigned to them. They may even have groups that have names that make reasonable sense to the creators of the folder. This is a workaround where the containing object is assigned the permissions and this is used to control access to the file. In small way this is utilising the only bit of metadata that is available, the path, to provide access control flexibility.

While not the only solution to more advanced access control, metadata file systems provide a unique solution to access control. In a metadata file system there is more than just the basic details of a file that most file systems record. A metadata file system doesn't just contain the name of the file, how big the file is, who owns it, what group is assigned to it and anywhere between 32 bits and 64 kilobytes worth of access control data - plus what ever other small attributes that exist which usually take up one bit each. Instead it can contain more details about a file in the file system layer. It can also start to build relationships between files. Instead of a file just existing on it's own, it can be linked to different objects. Metadata

file systems can easily store objects of pure metadata which may not actually have a file portion of them — or even generate a data stream to match the metadata stored transparently. In a metadata file system, a contact is a file in it's own right represented within the file system. Concepts such as tagging can be embedded directly into the file system without the need for it to exist outside.

Having this amount of data available within the file system also means that access control at the file system level can be rethought. Instead of creating access control rules for each file or even for folders at specific levels, more generic access control rules can be generated based upon the metadata. To provide this functionality a language is required to adequately encode the rules. XACML provides a standard language for defining the rules and constraints. XACML (see 3.6.6 and 5.2) provides the ability to define access control rules

File management in a university provides good examples of how this information could be used. Lecturers of a course should be able to edit material associated with a course they are teaching. A lecturer's user object is linked to the the course which is in turned linked to the course materials for the course. An access control rule could be constructed that permits lecturers to edit course materials that they are teaching. This concept can be expanded beyond course materials into student management (you can only see details of students who are enrolled in the courses you are teaching this semester) or assignment management (a student is linked to a course; a course is linked to an assignment; students can submit to those assignments which are linked into the system as submissions; lecturers related to a course can view submissions). In each of these examples metadata and relationships are used to provide the basis of access control.

Through use of attributes stored within the system in a uniform manner, a more fluid method of access control can be constructed. This also permits the use of an access control language over an increasing amount of data that is being stored within modern information systems. Data that already exists is being reused for example who is teaching which course, what assignments does the course have and course enrolments. This shifts the burden of access control away from discretionary rules and onto a point where access can be controlled through proper maintenance of existing metadata and general purpose rules.

In this way metadata becomes an important tool in providing flexible and powerful access control. With the advent of various search systems, the quality of data is increasingly being exposed. Data that is incorrectly catalogued or tagged with metadata appears visibly wrong and data with appropriate metadata is made easier to find.

A key issue in all of this is ensuring that items have sufficient metadata associated with them to make this form of access control work. In structured environments, such as a University, metadata association can be included as part of a position and training provided to ensure quality and accuracy of data. How this would work in a less structured environment would require further research and user testing.

6.2 Metadata Filesystem Background

MDFS comes at an interesting point in file system development. For the better part of two decades Microsoft has been promising an object relational filesystem but fails to deliver. The latest incarnation of this is "WinFS", a project slated to be Longhorn's file system (now released as Vista), was cancelled at an early public beta stage. It is in this context that we build MDFS. The lack of metadata capable filesystems, and in the case of MDFS and WinFS, strongly typed metadata filesystems, has created a development in 'indexing' data.

So parallel to file system developments, the area of 'indexing' and 'search' has be-

come far more apparent to now be an advertised feature of operating systems. This is true for both Apple's Mac OS X, utilising an indexing engine called "Spotlight", and Microsoft's Windows Vista with the creatively named "Windows Indexing Service" and "Windows Search". Both of these hold a prominent place on the desktop (top right corner click button and bottom left corner click button respectively) showing that information retrieval on the desktop is an important issue for both of these organisations. Linux also provides its own indexing and search system called "Beagle". Beagle is on par with the existing services such as Spotlight and Windows Search. In SUSE systems, Beagle is located in the "computer" menu. This way of accessing Beagle similar to the way Windows Vista operates.

While all of this has been going on, there has been the increasing use and awareness of the availability of metadata in existing file systems. While all major operating systems (e.g. Windows, Mac and Linux) have file system level support for metadata, very few applications actually make use of this facility. Within Linux, extended attributes are typically disabled by default for file systems, even if they support it. NTFS supports extended attributes although no part of the Windows operating system appears to use them, nor does any application on the platform. It appears the support for extended attributes in NTFS was designed with OS/2 compatibility in mind. With the release of Mac OS X 10.5, Apple have started to use the extended attribute support that was added in version 10.4 of their operating system. Apple is starting to use extended attributes to provide information about files such as where it was downloaded from. Additionally they provide a tool, xattr, that allows for easy access via the shell prompt to metadata stored on the filesystem.

Historically both OS/2, BeOS and Mac OS have provided extended attributes or features similar to it. Both OS/2 and BeOS supported strong metadata file systems (HPFS or High Performance File System and BeFS respectively) whereas Mac OS has utilised the "forks" approach to provide extra metadata information (e.g. a fork could contain much metadata about an application such as its dialogs). Forks are also supported under NTFS as "streams". A fork is basically a separate bit stream for a file entry, so a file might have multiple streams. Mac OS only supports two forks, a resource fork which can contain typed metadata and a data fork which is the traditional bit stream. NTFS can support multiple named streams, accessible by suffixing a ":" in the command line to access each stream.

With the beginnings of heavy use of metadata within modern day operating systems the next development is to provide logical storage capabilities of extended metadata information. The move to utilise a more structured manner replaces the current heavily unstructured methods. The extended attribute systems for all platforms only store plain strings without any type information. The aim of MDFS is to build a filesystem that handles attributes in a type aware manner while still being compatible with existing systems that might not be metadata aware.

The main aim of this is to promote the file system as capable of storing the information that has previously been made a part of the files that were stored on the filesystem. Good examples of this are MP3 (with its ID3 tags) or JPEG (with its EXIF image data). In both of these cases metadata about the file are stored in the file, instead of metadata about the file being stored in the file system. This has lead to other issues such as the need to index data to provide search solutions or database powered systems (Windows Future Storage, WinFS).

Using a database to manage file system metadata enables us to readily make use of database concepts to provide corresponding features in the filesystem. With in a database, the base construct is a table that stores rows and columns of information (tuples). Furthermore within databases 'views' are a common way of pulling information together from potentially multiple sources and filtering it but making it appear like it is a real "table." A view may also have certain criteria applied to it that means that it may not display all available data, for example it may limit the

number of rows returned or require that a certain field have a set value or a range of values. Additionally a database is typically very fast at compiling and executing complex queries on tables and can maintain indexes to ensure searches for data is fast.

What makes this useful is that when a filesystem stores its metadata and extended attributes within a database, it becomes very easy to search for those files that match criteria. When data like file genre is moved into a database then it becomes very fast for the filesystem to provide search results that are useful. Present operating systems provide similar functionality however instead of storing the metadata with the filesystem, it is being stored into another database on the filesystem, disparate from the files that it describes. Examples of this include Windows' search system or Mac OS X's Spotlight system. This makes it hard to present different 'views' of the filesystem to the user as it requires the development of specialised programs to view the data and extract it. Within a database powered filesystem, virtual folders can be created from database views to present the user with information. Within MDFS normal file and folder operations work on these virtual folders transparently. This means that the user or a third party developer doesn't have to do anything to utilise this support from within MDFS, it is exposed through traditional interfaces.

6.2.1 MDFS Purpose

The purpose of MDFS is to develop an implementation of a metadata file system to examine concepts and issues relating to metadata file system. Issues relating to metadata file systems involve how it is accessed, how it behaves and if there are any complications in implementation. The concepts covered involve inheritance of files within an object orientated hierarchy, permissions in metadata file systems and creating relationships between objects in a metadata file system. Examination and development of the properties found in metadata file systems and not other systems such as unique identifier and relationships between the items.

It is important that the system where possible is backwards compatible with traditional filesystems. This is to enable compatibility with existing applications so that custom support for the file system isn't required. A failing of earlier metadata file systems is the requirement to implement a specific API to even use the file system which hinders adoption of the file system. MDFS aims to demonstrate how metadata file system concepts can be examined whilst using existing tools to manipulate data. This also extends to implementing standard extended attribute support so that the extra fields are accessible via these methods as well.

Chapter 7

MDFS Specification

7.1 Requirements

MDFS is required to complete the following high level goals:

- be compatible with regular applications (e.g. vim, OpenOffice, etc)
- have extensible metadata schema with custom classes
- the ability to create smart folders that are equivalent to a traditional database view of the data stored in the database

The first requirement is the effective implementation of POSIX standard calls within the file system so that the host operating system can operate on the files in a standardised manner. The second is the development of the metadata aspect of the file system and the ability to create a flexible metadata schema for storing objects. The final requirement is a demonstration of the flexibility in the system to be able to utilise metadata and expose useful information to the user.

7.2 Use Cases

Within MDFS the key use case is depicted in Figure 7.1. MDFS is firstly a file system. So standard file system operations apply: create file, edit file and delete file. MDFS has a different view on folders than a traditional system however their structure can be emulated within a particular class structure.

In addition to support of the default file system management tasks that a file system should complete, MDFS needs to be able to handle extensible metadata classes (create classes) and utilise views upon the information stored (create views). Both of these operations are completely new to the file system uses as neither concept particularly exists within present day file systems. The use of "views" can be considered similar to "smart folders" found within Mac OS X which are basically searches. The last use of the file system is the assignment of metadata. As the file system is constructed on the basis of relying on files having extra metadata, the task of assigning metadata to files becomes an important use for the system and almost critical to it's operation.

The creation of views and classes is the domain of both the user and the developer. The end user of the file system should be able to create both of these elements and the original developer of the system will wish to create some classes for the maintenance of the system. Intelligent agents might wish to add files and also maintain metadata. One such agent could be a transducer which examines files and automatically extracts metadata and populates relevant fields.

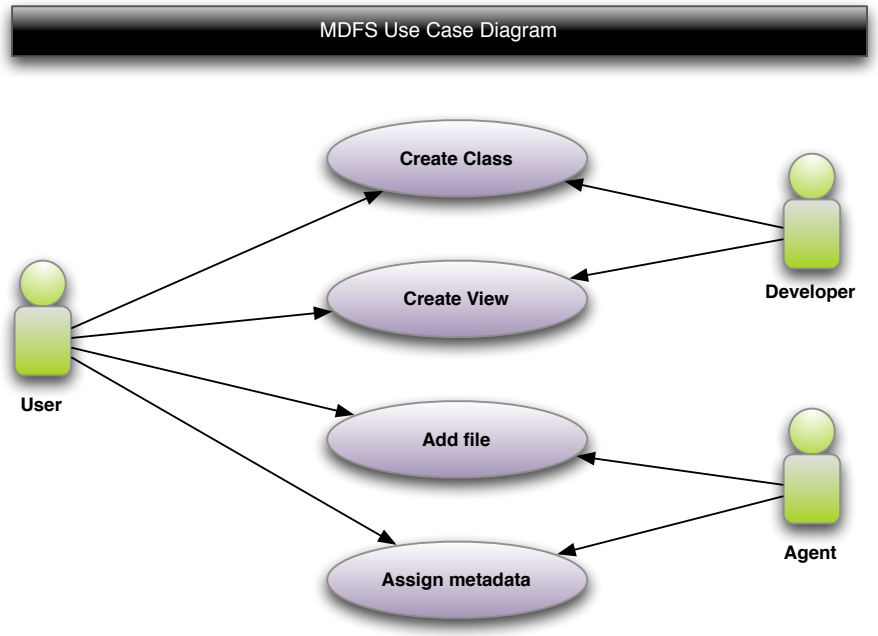

Figure 7.1: MDFS Use Case

7.3 Domain and Range

MDFS is expected to operate within the bounds of a standard filesystem, and as such should be capable of storing files. Additionally MDFS is designed to allow for the creation of custom metadata classes and provide the functionality to add, edit, retrieve or destroy metadata associated with individual objects stored within the system.

It is envisaged that the system would allow the creation of pure metadata objects, objects that do not have any file data associated with them. For example a JPEG image might be considered to be composed of two parts: the raw image data and the metadata about the image (such as EXIF data that describes the particular camera that took the picture). Other objects are more pure structured data or semi structured data, such as that of a contact. A contact deviates from the fine definition of structured data due to its inherent flexible nature. MDFS is not capable of providing a method of manipulating semi structured data however it does provide the capability to build structured metadata storage for files and pure metadata objects.

MDFS exposes three root folders by default:

- admin
- classes
- views

Each of these folders shows different perspectives of the system. The admin directory for example shows information about the system and allows the creation, modification and deletion of both classes and views. For anyone who is familiar with the proc file system under Linux this will appear familiar.

The classes and views directory both show perspectives about files stored within the system. Classes shows a listing of all available classes as directories and then within those directories shows the member objects of those classes.

Views is analogous to a saved search in Apple's Finder file manager and Spotlight indexing system. A saved search, or "Smart folder", is run automatically when ever the folder is viewed by the system. The MDFS views behave in a similar way to these virtual folders except they appear as a real part of the file system, unlike the Smart folders. An example of the key difference is when a Smart Folder is accessed via a third party application (e.g. NeoOffice, a OpenOffice.org port) it does not behave consistently as a folder, however when using other applications is does. The aim of MDFS is to build into the file system core so as to provide a seamless user experience for all applications.

7.4 Metadata System API

The metadata system is what distinguishes MDFS from many other file systems. This API provides the ability to manipulate the metadata storage system for objects.

7.4.1 View specification

A view within MDFS is similar to a view within a database however the system handles extra information associated with the view to allow a smoother operation.

- `createView(char * viewName, char * viewSpecification, char * fileKey)`
  This creates a new view with a given view name and SQL specification. The fileKey is the key specification for the filename. This returns true or false depending on the success of the operation.
- `deleteView(char * viewName)`
  Deletes a given named view.
- `alterView(char * viewName, char * viewSpecification, char * fileKey)`
  Alters a given named view and changes it to be the specification. The fileKey is the key specification for the filename.

7.4.2 Class Specification

A class is the container for all objects with in the system. MDFS as a system classifies objects within the class hierarchy. Classes form the basis of all stored objects and relationships within the system. Unlike views, classes cannot be altered via this interface due to the impact it might have upon other subclasses or stored metadata.

- `defineClass(char * className, char * classSpecification, char * fileKey)`
  This defines a new class with the name and SQL class specification. The specification should inherit from the default Fileable or Non-Fileable classes or their children however the system will not directly enforce this restriction.
- `removeClass(char * className)`
  This removes a leaf class from the system. All members of the class are automatically moved into its parent class. The three special classes Base, Fileable and Non-Fileable cannot be removed. This operation may take some period of time to complete depending on the number of members in the class. Only classes without any children can be removed.

Figure 7.2: View or Class Definition Format

```
# Filename
# Key Value
SQL Query definition of view or class (create table or create view)
```

Figure 7.3: View or Class Definition Format

7.5 Input and Output Formats

MDFS behaves like a file system and as such its input and output is handled in a manner that a filesystem should behave as per POSIX specifications and through the FUSE API. This includes both the normal operations and the use of the extended attributes in the system. However there are some aspects of the file system that are purely virtual and determined by the file system itself. The primary user interface into the system is through the file system layer of their operating system and the configuration of the bootstrap program.

7.6 View and Class Interface Format

The views of the file system are presented in the "views" virtual directory as directories that the user can navigate to with the results of the view available as files. The classes available in the file system are presented in the "classes" virtual directory as directories that the user can navigate to with the members of the class available as files. The final directory is the admin one which consists of "views" and "classes" each containing the definitions for both the "views" and "classes". The views are updateable via this interface, however the classes are not. Irrespective of this classes may be removed as per normal (see documentation on the removeClass operation for valid removal conditions) and both views and classes may be created by defining a new file with the relevant data in it.

The format for this file is described in Figure 7.3.

Chapter 8

MDFS Design

8.1 MDFS Architecture

What makes MDFS different from previous efforts is that it doesn't provide simple generic metadata access of the sort that is available presently within file systems NTFS and ext3. MDFS instead provides *typed* and *controlled* metadata, like a database system. This is the key difference between this system and existing file system solutions.

8.2 Functions and Algorithms

This section documents the functions and algorithms required for the correct operation of the proposed metadata filesystem.

8.2.1 FUSE API Documentation

MDFS implements features exposed from the FUSE API in conformance with the POSIX API. As such the following functions from FUSE are implemented. This means all of the regular file operations (such as open, close, read, write, stat) are implemented but additionally the extended attributes are also implemented as the interface into the metadata file system (getxattr, setxattr, listxattr).

Within each function call also exists a set of routing functions that handle the direction of different virtual folders to handle operations in a logical manner. As such a generic switcher would look similar to this:

```
FUNCTION genericFunction(path, otherparams)
        IF path STARTS WITH 'views' THEN
                useViewHandler(path, otherparms)
        ELSE IF path START WITH 'classes' THEN
                useClassHandler(path, otherparams)
        ELSE IF path STARTS WITH 'admin' THEN
                useAdminHandler(path, otherparams)
        ELSE IF path IS '/' THEN
                useRootHandler(path, otherparams)
        RETURN FILE_NOT_FOUND
END FUNCTION
```

Figure 8.1: Routing Example

This is the generalised first level for all filesystem functions. From here the specifics of each operation are handled.

There are multiple classes of functions for the implementation that designate the behaviour when compared to a traditional file systems counterparts. This may be related to database issues or otherwise.

All functions use the signature defined by the FUSE API, as such it is not noted here. For more information on the signatures of these functions please consult the FUSE API or the fuse.h file and examine the "fuse_operations" structure.

Directory Permission and Operations Notes

Different directories may or may not be writable depending on how they are generated within the system. For example the directories that are built from the solid classes are writable because their generation is actually through a simple select. The same is also true for any view that contains a concrete specification and does not use ranges (less than or greater than for example) or include elements that would otherwise be immutable or is defined by the file system. For example attempting to create a file in a view which is defined by a constraint based on file size will return an operation not permitted error. This applies to any operation that attempts to create a new object within the system (linking, moving or object creation). As the keys are unique within the system, it is possible to open and update an object in a view at any point, there are only limitations applied to attempting to add objects to the view.

Standard Functions

These are functions that behave and return normally without any changes from the perspective of the user. Any typically valid operation should be permitted and operated without issue. All functions should behave without any side effects (as per most file systems intended implementation). Some operations may return errors as specified by section 4.1.1 Directory Permission and Operations Notes. For more information about specific functions, please see Appendix 2.

- `getattr` and `fgetattr` (similar to stat)
- `read, open, write` and `release` (similar to a close)
- `lock`
  A lock is created on a particular file. This is also stored within the metadata system to allow for contiguous links.
- `truncate` and `ftruncate`
  These operations are passed through to the underlying file system. Their return value is condition on the result of that operation. These operations are not valid on objects that do not store raw data (e.g. objects that do not descend from the Fileable class). If ftruncate is not implemented then truncate will be used instead.
- `utimes` and `utimens`
- `fsync`
- `access`
- `create` or `mknod`
- `chmod` and `chown`

- `opendir, readdir, releasedir, fsyncdir`
  These commands all return normally except for fsyncdir in the case of where a directory has been created (somehow). The fsyncdir will then return an error.

- `destroy`
  The destroy operation disconnects from the database and attempts to commit any changes that need to be made.

Unimplemented Functions

- `readlink`
  All links within the system are presented as 'real' members of the filesystem (see below comments on link and symlink for more information). As such there are no links for the system to read.

Prohibited Functions

- `mkdir`
  MDFS does not use directories in the conventional sense so it is impossible to create a new directories as all directory structures are maintained through the implementation of classes or views.

Functions with Altered Behaviour

- `link` and `symlink`
  A link and symlink behave in identically, as such under MDFS it is impossible to distinguish between a hard link or a symbolic link. Within MDFS when these are called on locations that are write-able the system will attempt to alter the metadata for the object so that it matches that of the new destination. Where this is possible these functions will return a success message, however in cases where the metadata cannot be updated appropriately an operation not permitted error will be returned. An example of an illegal link would be linking from one class to another as MDFS enforces single inheritance and membership for object classes. Links between views are possible if the target view allows updates to be made to the view. In the underlying database, views might be made up using equality operators or through the use of inequality operators (e.g. greater than). If a view is made up from equality operators then the contents of the view can be altered as the result of the operation will always be defined by the constraints of the view. If a view includes an inequality operator then a range of possible solutions could match so this form of view is not able to be altered.

- `unlink`
  Unlink is similar to link in that instead of changing pointers, the metadata that makes an object a member of a particular view where this is a valid operation or alternatively an operation not permitted error.

- `statfs`
  The statfs call returns file system statistics. This includes both the database usage and the physical disk used to store files.

- `setxattr` and `getxattr`
  These functions only work for the fixed attributes of a given files class. They will return an operation not permitted error when trying to set or get attributes that are not valid for the class of the object being queried. This call is bounded by the database constraints which may also cause the call to fail.

```
Base Table
objectid                bigint (PRIMARY KEY)
createddate             timestamp
modifieddate            timestamp
owner                   integer
group                   integer
permissions             integer DEFAULT 755
symbolicname            character varying(255)
symbolicextension       character varying(255)
locked                  boolean DEFAULT false

Fileable Table extends Base Table
inode                   text
hash                    varchar(128)
size                    integer

Nonfileable Table extends Base Table
```

Figure 8.2: MDFS Core Classes

- `removexattr`
  This function does not remove the attribute however it does set the value of the attribute to be that of the valid null value where possible or an error if it is not possible to have a null value in that field. This call is bounded by the database constraints which may also cause the call to fail.

- `rename`
  This operation may not be permitted on objects where the file name is a composite value or is immutable by the database (sequential database key). In cases where the filename is designated by a single field, a rename operation will update this field value to be a new value within the constraints of the database.

8.2.2 Database Classes

MDFS is primarily composed of the classes and views in the PostgreSQL database. There are also some special support classes. In this section tables may be used in place of classes,.

Core Classes

There are three Core classes, the Base class and its children the Fileable and Nonfileable classes. The class system uses inheritance so that children inherit any attributes of its parents. Classes are implemented as stand alone tables in the PostgreSQL database. See Figure 8.2.2 for their implementation details.

The Nonfileable table has no members of its own it is for the system's purposes when determining what action to complete and to keep the top level structure clean.

System Support Tables

The system utilises support tables to enable additional system data to be stored on top of the existing metadata.

The relationship table is used to enable simple object ID relationship tagging. Its attributes are two object identifiers and the value of the tag. Deeper SQL relations

```
MDFS_Relationships Table extends Base Table
oid1    bigint
oid2    bigint
value   varchar(255)
```

Figure 8.3: MDFS Relationship Table

```
MDFS_Classes Table extends Base Table
classname       varchar(255)
filename        varchar(255)
```

Figure 8.4: MDFS Classes Table

are specified within the database provided features. This is to provide object to object instance relationships as relationship types are not defined within this system. The relationships implemented in this system are analogous to a tagging system.

The classes table is used to provide extra information about building views and classes. The classname is the name of the class that the entry describes and the filename is a list of attributes (prefixed by a dollar sign ($)). By default the filename is the object ID of the instance (as this is available for all objects).

8.3 Scenarios

In this section a scenario around a student completing assignments is developed. The situation examines the use of features of file system to demarcate works for different courses. This includes assignments that were submitted, course materials and other notes. In developing this use case, I will derive this from an existing system where the file system path is used to encode this metadata.

8.3.1 Existing System

The current system has a folder with each year studied listed within it. Each year lists each semester studied and within each semester the courses are listed by their course code. Beyond this layer the path has no further semantic meaning however for the purposes of the scenario all of the materials included are a part of an assignment. A graphical depiction of this is demonstrated in Figure 8.5.

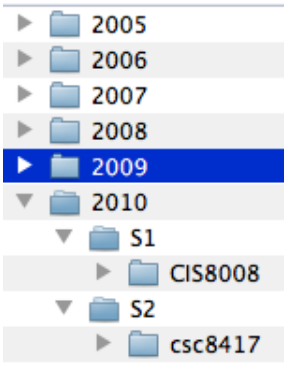

Figure 8.5: Current layout of courses

The system as it currently exists works well during the active semester because

```
/2009/S1/msc8001
 ^    ^  ^
 |    |  Course Code
 |    Semester
 Year
```

Figure 8.6: Path structure

the structure of the system is clearly delineated by time. If I know when I studied a particular course it is very easy to find the relevant submissions for the course. However there are two use cases which this structure makes hard:

- Finding a particular course quickly regardless of when I studied it (e.g. I know the course code but not the year or semester).
- Finding the contents of a particular course quickly regardless of when I studied it or if I studied it multiple times.
- Finding a course by a name not a code.

The first problem is solved by a brute force traversal to find the relevant folder name or through the use of an external indexing service that has already completed the traversal. The second problem is a bit trickier. The last problem is much harder to complete without complicating the structure. The folder structure could be altered to include the course name however this then complicates directory traversal in command line environments.

However the path structure is being used to encode three disparate fields as depicted in Figure 8.6. Each of these are actually structured information stored within the path. The requirement to encode this information into the path is a limitation of the existing file systems: file names and their paths are the only metadata for a particular file.

8.3.2 MDFS System

In the MDFS system, a class called "Assignment" would be developed. This class will derive from the base "Fileable" class and will store files as well as the metadata associated with them. Instead of encoding the information into the path, the information would be encoded in the table. The advantage of doing this is that any form of structure could conceivably created. The present structure (year, semester, coursecode) could be created in addition to potentially alternative structures (e.g. coursecode, year, semester or even coursecode, year-semester). The problems identified with the previous structure could easily be solved.

The first problem of finding a particular course is easily solved either by creating a view based on alternate structures. One such structure could have the root based on course code with year and semester studied underneath. Views could be constructed to match those courses as well. This would also resolve the second point of finding the contents of a course regardless of when it was studied.

The last problem is an interesting situation. In this case a secondary Non-Fileable class called "Courses" could be related. The coursecode, year and semester information could be shifted from the "Assignment" class and references made between the file and the course instance. This would result in a more normalised design and allow for extra fields that could be used to search for items using a more extensive view.

In this particular scenario, the use of a structured backend makes effective retrieval of information much easier to complete over existing scenarios. The extra

data and the ability to extend the captured information provides for more flexibility and information retrieval scenarios.

Chapter 9

MDFS Implementation

As a part of this study of file systems a metadata file system has been created. Metadata file systems might be thought to represent the next generation of file system development just as hierarchical file systems replaced record based systems from early mainframes. This chapter presents a view of metadata in a file system context, methods of enhancing already rich objects in the system.

9.1 Base Infrastructure

9.1.1 FUSE

File system in User space (FUSE) is a is a simple API that enables the simple development of file systems. FUSE is useful because it allows the entire application development to occur in user space without having to require the development of potentially dangerous kernel modules to complete the project. This means that a more rapid development approach can be taken as well as the ability to take full advantage of everything that is offered in a user space system (such as database drivers).

The FUSE API is available natively from Linux and an equivalent called "Mac-FUSE" is available on Mac OS X. As the VFS semantics between the two differ slightly, they are not entirely compatible but are reasonably compatible for most functions. The pathway of a FUSE call is depicted in Figure 9.1.

9.1.2 Base Design

The base part of the system is the implementation of a plugin for the FUSE system. As FUSE is aimed at presenting information through the virtual filesystem layer of the kernel, the output of MDFS will mimic that of a filesystem and will be mounted in a way similar to a network mount.

From FUSE, MDFS will connect with both PostgreSQL and the host file system. For file operations the system will proxy most operations on fileable classes onto the host file system where the data is stored. For metadata manipulation MDFS will utilise the POSIX xattr functions (getxattr, setxattr, listxattr, remotexattr) which avoids having to write a new API for filesystem interaction with the metadata system.

On the metadata side the system is made up of objects and views of those objects. An object is something that stores metadata (a pointer to a bit stream is just another part of the metadata) that is handled by the system. The metadata is stored within the PostgreSQL database and for those objects in the fileable class they will be stored within the file store on the host's filesystem. The file system

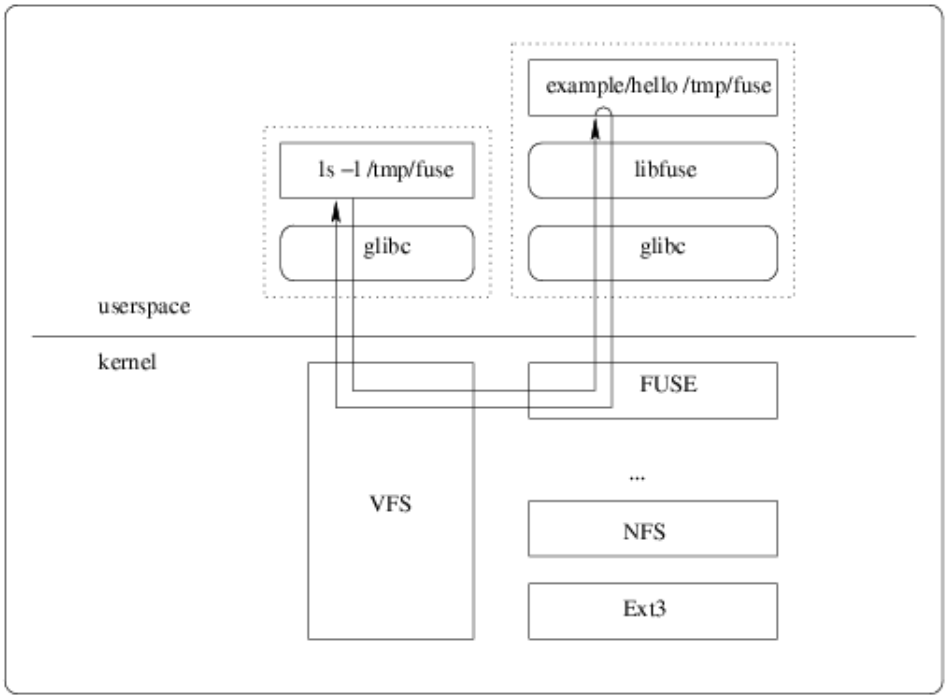

Figure 9.1: FUSE Structure [5]

should provide the ability to interact with existing utilities without requiring their alteration. While applications can feature the ability to utilise rich metadata the file system should behave normally when unaware applications are utilising the system.

9.2 MDFS Implementation

MDFS is a MacFUSE/FUSE API compatible user space file system that has been developed primarily in C. MDFS uses PostgreSQL to store the metadata aspects of the filesystem. PostgreSQL was chosen because of the full implementation of object-orientation, including inheritance. These features offered by PostgreSQL are not available in comparable open source or free database systems. For blob storage of actual file data, MDFS defers to the parent file system and passes most traditional file operations onto the host operating system and it's preferred file storage system.

MDFS emulates the UNIX permission model through using owner and group fields in addition to a permissions field that provides storage for the traditional four digital octal mask for file permissions. This permits full compatibility with ownership operations and the fields are also exposed via the standard extended attribute API for setting and retrieving these attributes.

Through PostgreSQL, MDFS creates object classes through distinct table types. Tables derive from a base table and views contain at least as many fields as is defined in the base table (e.g. symbolicname, symbolicextension). PostgreSQL's unique support for object orientated style hierarchies makes it ideally suited to replicating an object tree within a database context. This feature also implicitly includes child items of the tree into the parent tables however the extended attributes (columns) are unavailable in the parent views. Figure 9.3 demonstrates a class model that is then replicated in PostgreSQL as a set of tables.

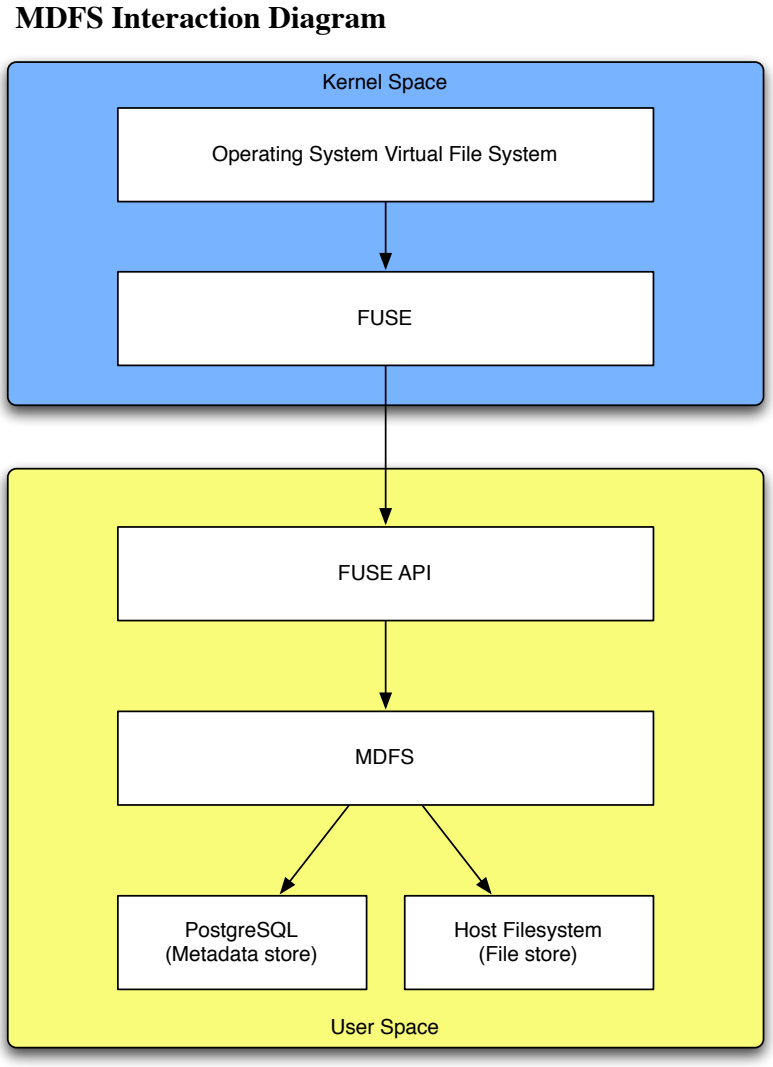

Figure 9.2: MDFS Interaction Diagram

As demonstrated in Figure 9.3, there are classes which define distinct relationships (e.g. user defines "studies" and "submits" relationships to course and assignment respectively). Some classes are mostly empty and exist as named holders to be included in subclasses. When fields are inherited from their parents they don't need to be recreated at the child level. The ability to use relationships as a native part of the file system comes naturally from the relational aspect of the underlying PostgreSQL database.

Another feature that is useful with the database setup with PostgreSQL is the use of sequences to generate unique object identifiers within the file system. This means that any object at any level of the file system is uniquely identifiable regardless of where it is located within the system. This enables any object to be linked easily to any other object.

MDFS also provides basic support for views within the file system and they are presented simply as folders. All views within MDFS need to support the minimum fields defined in Base but can feature extra fields relevant to the view. Views cannot be nested within other views under the present implementation.

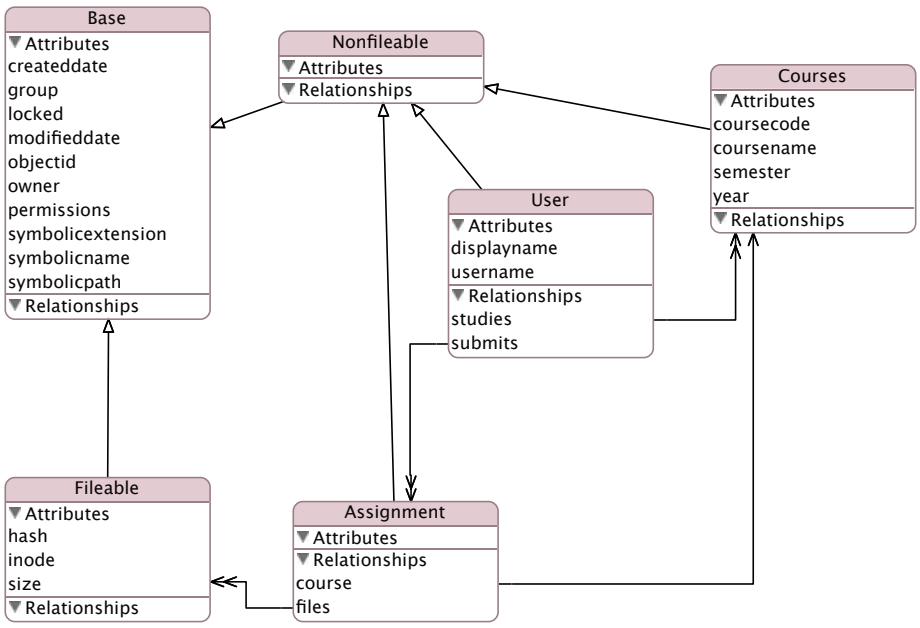

Figure 9.3: MDFS Class Hierarchy

9.3 Prototype

The MDFS prototype system is designed to show how a metadata filesystem might be implemented within the contexts and bounds of a traditional filesystem. As such MDFS has been implemented using the interfaces already exposed by POSIX to ease integration of the product into an existing environment without the need for extensive rewriting and complication of functionality by using a special purpose API to access metadata, which was the approach of WinFS. MDFS further deviates from the WinFS model by allowing database classes to be created without the need for the creation of any specific programming unit. The definition of this class is simply that of a standard SQL creation statement.

9.4 Environment

MDFS is written in C and utilises the FUSE API to provide integration with the filesystem and PostgreSQL to provide metadata storage. MDFS has been tested on the following platforms:

- Mac OS X 10.4.10+:
  - PostgreSQL 8.2.3 supplied by Marc Liyanage
  - FUSE API supplied by the MacFUSE project (release 0.4.0)
- Debian Etch
  - Standard Debian packages for both FUSE and PostgreSQL

The base requirements for any untested platform should be:

- PostgreSQL 8.2 or greater
- FUSE API 2.6

- A parent file system

At the present moment there is no FUSE implementation for Windows operating systems which means that there is no way to support MDFS under Windows.

MDFS does not require any explicit hardware requirements to operate, however due to its nature it is suggested that where possible the host machine have as much RAM as possible to ease multiple operations and caching.

9.5 Operating Instructions

MDFS operates as a filesystem however due to its nature as a FUSE powered file system it does not behave in quite the same way as a kernel-level filesystem.

You can start MDFS by using the following command: `/path/to/mdfs_executable /path/to/mountpoint` This will connect to the default PostgreSQL database and default MDFS data store.

Configuration is drawn from either "/etc/mdfs/config" or " /.mdfs/config". The configuration file is an INI formatted file with the section headings match the desired mount path.

9.6 Experimental Scenarios

Mounting the file system is trivially completed through running the MDFS command with the target mount point:

```
silversaviour:Debug pasamio$ ./mdfs test
silversaviour:Debug pasamio$
```

Navigating into the directory, the MDFS root folder structure as defined in 7.3:

```
silversaviour:Debug pasamio$ cd test
silversaviour:test pasamio$ ls
admin   classes views
silversaviour:test pasamio$
```

At the present moment the "admin" function isn't fully implemented and listing the directory results in a blank operation.

9.6.1 The Classes Directory

The "classes" directory lists available classes in the system:

```
silversaviour:test pasamio$ ls classes
Assignments             Fileable                Nonfileable
Base                    MDFS_Relationships      Users
Courses                 Media                   Video
silversaviour:test pasamio$
```

In the list we can see the "Base", "Fileable" and "Nonfileable" classes. The "MDFS_Relationships" class is also listed from 8.2.2. Other classes that are available in this instance are "Assignments", "Courses", "Media", "Users" and "Video".

While not readily apparent within the file system view, the classes for the system are structured in the hierarchy listed in Figure 9.4.

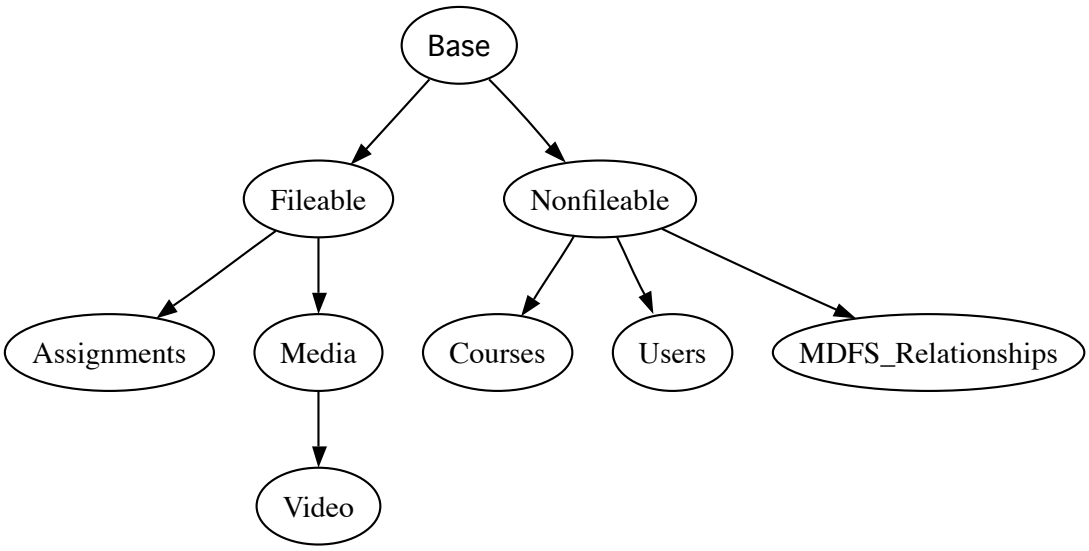

Figure 9.4: MDFS Demonstration Class Hierarchy

9.6.2 The Views Directory

Having a look at the "views", we can see only two views:

```
silversaviour:test pasamio$ ls views
CSC3414         Portfolio
silversaviour:test pasamio$
```

The views being available are "CSC3414" and "Portfolio". CSC3414 only has one file within it:

```
silversaviour:test pasamio$ ls views/CSC3414/
index2.php
silversaviour:test pasamio$ head views/CSC3414/index2.php
<?php
/**
* @version              $Id: index2.php 9764 2007-12-30 07:48:11Z ircmaxell $
* @package              Joomla
* @copyright    Copyright (C) 2005 - 2008 Open Source Matters. All rights reserved.
* @license              GNU/GPL, see LICENSE.php
* Joomla! is free software. This version may have been modified pursuant
* to the GNU General Public License, and as distributed it includes or
* is derivative of works licensed under the GNU General Public License or
* other free or open source software licenses.
silversaviour:test pasamio$
```

The CSC3414 view is defined as the following:

```
CREATE OR REPLACE VIEW "CSC3414" AS
 SELECT         "Assignments".objectid,
                "Assignments".createddate,
                "Assignments".modifieddate,
                "Assignments".inode,
                "Assignments".hash,
                "Assignments".size,
```

```
                "Assignments"."owner",
                "Assignments"."group",
                "Assignments".permissions,
                "Assignments".symbolicname,
                "Assignments".symbolicextension,
                "Assignments".locked,
                "Assignments".coursecode,
                "Assignments".semester,
                "Assignments"."year"
   FROM "Assignments"
  WHERE lower("Assignments".coursecode::text) = 'csc3414'::text;
```

The Portfolio view has a few more files listed within it:

```
silversaviour:test pasamio$ ls views/Portfolio
CSC34152.txt                    index.php
CSC3415_Assignment_Demo.txt     index2.php
CSC3415_Evaluation.txt
silversaviour:test pasamio$
```

Portfolio is defined as the following:

```
CREATE OR REPLACE VIEW "Portfolio" AS
 SELECT         "Assignments".objectid,
                "Assignments".createddate,
                "Assignments".modifieddate,
                "Assignments".inode,
                "Assignments".hash,
                "Assignments".size,
                "Assignments"."owner",
                "Assignments"."group",
                "Assignments".permissions,
                "Assignments".symbolicname,
                "Assignments".symbolicextension,
                "Assignments".locked,
                "Assignments".coursecode,
                "Assignments".semester,
                "Assignments"."year"
   FROM "Assignments"
  WHERE         lower("Assignments".coursecode::text) ~~ 'csc3415%'::text
        OR      lower("Assignments".coursecode::text) ~~ 'csc3414%'::text;
```

9.6.3 Handling Metadata

An important aspect of the file system was the ability for the metadata in the system to be exposed via standard API's available on the host operating system. MDFS exposes all of the attributes of a particular class in order. Building on the Portfolio view listed in 9.6.2, this is the output of a sample file included within the view:

```
silversaviour:Portfolio pasamio$ xattr -l CSC3415_Assignment_Demo.txt
CSC3415_Assignment_Demo.txt
                objectid       305
             createddate       2008-03-03 21:44:37.614859+10
            modifieddate       2008-03-03 21:44:37.614859+10
                   inode       /mdfs/store001/305.dat
```

```
                hash       <0000>
                size       0
               owner       501
               group       501
         permissions       777
        symbolicname       CSC3415_Assignment_Demo
    symbolicextension      txt
              locked       f
          coursecode       CSC3415
            semester       3
                year       2007
```

The command used is a standard extended attributes manipulation tool and displays the relevant information about the file. This includes the unique identifier (objectid), relevant modification dates, where the file is actually stored on disk (inode), the basic UNIX permissions support (owner, group and permissions) and the symbolic name and extension (symbolicname and symbolicextension respectively). The additional metadata fields are also included such as coursecode, semester and year.

The metadata can be manipulated using the standard commands. Altering the coursecode to be a different value will remove the entry from the view:

```
silversaviour:Portfolio pasamio$ ls
CSC34152.txt                    index.php
CSC3415_Assignment_Demo.txt     index2.php
```

As this is a weak view (the view is defined with an OR statement), the item needs to have its metadata updated in the relevant class for the file:

```
silversaviour:Assignments pasamio$ xattr -s coursecode MSC8001
        CSC3415_Assignment_Demo.txt
```

Then when the Portfolio directory is reviewed, the file is not included in the listing:

```
silversaviour:Portfolio pasamio$ ls
CSC34152.txt            CSC3415_Evaluation.txt
index.php               index2.php
```

9.6.4 File system Throughput

Due to the design of the file system, it appears that file system throughput is reasonably efficient. The system's file read and write infrastructure are direct proxies for their native file system counterparts and result in proxying the information. This means that there is a latency overhead as six context switches are required to complete a read operation. The requesting application does a read syscall on the file system, the kernel then takes over and routes the request back to FUSE, FUSE then switches back to user space to handle the request within MDFS, MDFS then translates using its internal file open table which then triggers a read again back into kernel space where at that point the underlying file system (usually a kernel module) will handle it and everything switches back. This is depicted in Figure 9.5.

The call structure adds another layer of context switches (6 switches instead of 2) however this doesn't appear to hamper performance. Figure 9.6 depicts a H.264 encoded video being streamed off an MDFS volume.

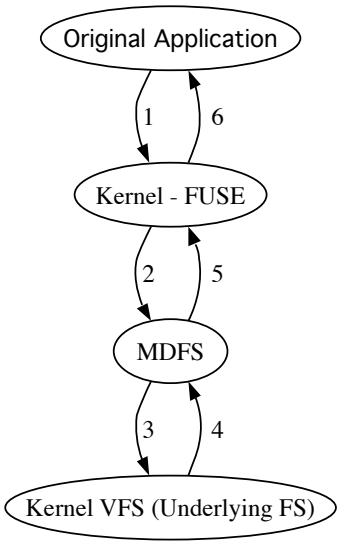

Figure 9.5: MDFS Context Switches

9.7 Access Control Model

The access control model for MDFS follows the standard POSIX model. The MDFS base class implements fields for an owner, a group and a octal permissions mask. This reflects the model used by Mac OS X and Linux. This was done to ensure maximum compatibility with existing file system technologies without requiring more advanced access control mechanisms to be enabled. At the present moment these fields are passed through the FUSE layer onto the host VFS layer which then uses the information to make access control choices. Much of the access control code in the file system at the moment is a reflection of this interface.

In developing my file system it became apparent that a language for specifying rules was required to take full advantage of the features offered by the file system. MDFS provided the basis of a system to store metadata about files and relationships between files but this on its own didn't provide access control. It becomes necessary to built a more expansive engine to evaluate rules as opposed to relying upon the built-in access control regime provided by the POSIX model. Such a system presents itself with the XACML model. XACML provides not only a language for expressing rules but also provides a model for handling access control. XACML provides the ability to define various rules and could be adapted to fit within the MDFS infrastructure. The verbosity and complexity of XACML does present an issue both in implementation and potentially in general use. However XACML has not been implemented in the file system at this point due to those complexity issues in implementation.

9.8 Further opportunities

The file system as designed works in combination of the FUSE API layer and the metadata storage layer in the PostgreSQL server. This design feature means that the MDFS system can easily be extended into a web accessible system relatively easy with similar features exposed through a web powered interface. This could be written using more web friendly languages than C such as PHP or Python. As the majority of the information is stored within a conventional PostgreSQL database,

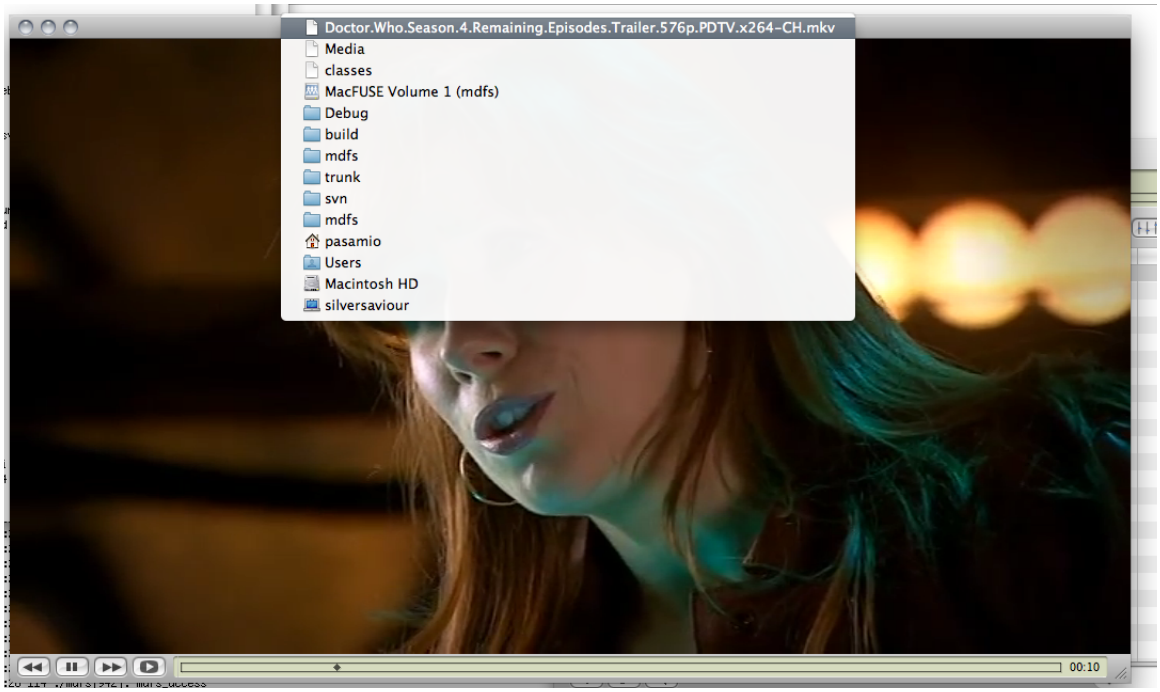

Figure 9.6: Sample H.264 576p Video Playing in VLC from MDFS volume

many of the features could be provided via a web console. This would improve the accessibility of the system and provides for interesting research prospects.

Part IV

Conclusion

Chapter 10

Concluding Remarks

10.1 Access Control Architecture Scenarios

Throughout the dissertation it has become apparent that access control systems are fragmented. This is perhaps most ideally depicted by the the disparate nature of access control in file systems. While the Windows model is slowly being refined into the dominant access control paradigm for file systems and related work, it also misses out on some of the features and properties introduced in other file systems.

What each of these systems also lacks is a uniform method of describing access control and transferring it between systems. When a file is taken from a NTFS volume and transferred onto any other volume that is not NTFS formatted, the ACL's and permissions aren't retained properly. The general lack of metadata portability is in itself another problem that needs to be solved, though I feel that there is a general purpose solution for the transmission security metadata is resolved through tools such as XACML.

10.1.1 Information Overload

While not explicitly identified in the main body of the work, information overload is quickly becoming a problem as increasing amounts of data is being generated. As data is generated, access control rules need to be associated to ensure that appropriate access to information is permitted. inappropriate access control can be bad at both extremes. Access control that is too lax and permits too much access is certainly not a good thing and will generally result in over disclosure of information to those parties who shouldn't have access to the data. At the opposite extreme, excessive security can result in those who need access to the information not being able to readily access the information presented. A potential solution is provided by increasing the semantic nature and associated metadata being applied which can enable access control rules to be written based upon the metadata and permit general rules to be written that apply to a wide variety of situations.

In a future system the prevalence of metadata and a flexible access control system will permit advanced access control that can easily scale out to extensive amounts of data. A demonstration system using a metadata file system which stores extra information about files can be used as a basis towards demonstrating how generic information systems with semantic information associated with them can be used to provide advanced access control.

10.1.2 XACML

XACML was covered in 3.6.6 and 5.2 as a XML based language to define access control. XACML has many properties where it is possible the emulate the way that file system permissions behave. On simpler permission models, such as the UNIX permission masks, the system is relatively easy to implement. The NTFS model demonstrated showed that XACML could reasonably emulate NTFS permissions even though the resulting code was rather verbose.

The use of XACML then provides the ability to have a uniform language for expressing access control in addition to defining a model for how an access control system can work. XACML also provides the ability when used with other XML formats to create access control rules using the contents of the documents. This provides the ability to create access control rules using the metadata available within the document to protect it in ways described elsewhere in this dissertation. If an image contained an XML portion which defined within itself those people tagged within it, perhaps utilising well known URI's for them in addition to other identifying features, then this information could be used by an access control implementation to provide advanced rules for access control to those documents.

10.2 Future Work

More work in this area could be seen in the form of ensuring that tools like XACML are integrated in such a way that this form of semantic access control is possible. XACML provides the strongest case for a uniform method of describing access control. While not a key component of the dissertation, there exists a distinct lack of portability for describing access control between systems. File systems mostly retain rules by transferring ACL's directly but this only works on compatible file systems. One cannot for example take an ACL from a Mac OS X HFS+ disk and then transfer it directly onto a NTFS formatted partition. Nor can one take a file from NTFS and shift it onto any other file system and expect that the ACL's defined for those items are retained.

10.2.1 Universal access control

So in future a uniform method of transferring access control rules could be seen as an advantage for ensuring that security is retained even through shifts with different media. Within the file system space this is perhaps less of an issue. Most of the time full local access is granted however increasingly information is being stored in what is referred to as the "cloud".

The cloud presents interesting challenges for various reasons however one concern is that of data availability and portability. Access control while not as deeply considered should be thought of as an important piece of this puzzle. Services like Facebook aren't typically considered in the cloud model but also form a part of cloud based systems. Other items like Google Docs provide a method of sharing documents from the cloud with access control as well. Documents exported from this system again have their access control stripped just as when the files were shifted between disparate file systems.

The only way to ideally solve this problem would be the introduction of universal and uniform access control, perhaps based on XACML, that could ensure that a document retained it's security throughout its lifetime. It needs to be recognised however that security is always only as secure as the weakest link. A user with physical access will usually be able to bypass most security systems for a computer. Enforcing true security then requires that documents are also encrypted to prevent unauthorised access without appropriate keys.

Parts of this exist within the trusted computing platforms from various systems. MULTICS enforced hardware protections via a ringed system. This system still exists today on modern CPU architectures however the number of rings is greatly diminished. The ring concept assumed that only certain points could be used to switch from an outer ring to an inner ring. All other accesses resulted in an access violation and the termination of the requesting process. Increasingly hardware level protections are being used for formats like HDMI to ensure that the signal being sent through different devices isn't intercepted and redirected to a storage device.

This level of hardware control provides an interesting perspective when considering access control for semantic information systems. The question of trust for the data and the source of the data becomes increasingly important in a world where previously metadata has been ignored for anything of importance. Increasing the reliance upon metadata in semantic systems for access control will also require greater care and extra information to be stored for the creator of the metadata — something not tracked at the moment. A requirement for metametadata (e..g metadata about the metadata) to track who assigned the metadata and when it was created. This needs to be explored as to how this metadata can be secured against malicious editing.

10.2.2 Social access control

Social access control is rethinking the how the subject of access control is approached. Most access control systems have ultimately been derived from military based standards or military funded work [75]. Examples of this are Windows which was certified to work with the Orange Book (covered in 1.2.4) and UNIX's access control model which derived from early work in MULTICS funded by the Department of Defence. The future of access control is mixed: in some situations the current approach to access control is still required and desired by large organisations and the defence establishment. However with the development of social networking and social network based systems this defence originated model is perhaps not the most applicable model.

Access control in social networking systems is much more fluid. Paul Adams, a user experience researcher at Google, recently made a presentation that observed that people in social environments have multiple groups that are distinct from each other [56]. Paul used the example of Facebook linking together distinct social groups, a group of LA friends and kids from a swimming club, with information that might not be appropriate for one of the groups. This form of access control is commonly referred to as "privacy control" but the function undertaken is controlling access to information.

It is my belief that more research into the development of social networking access control systems that automatically segregate different groups and utilises an intelligent agent approach to rule generation would enhance the overall privacy control and access within the social networking space. Unfortunately the present state of access control appears either excessively primitive (Google Buzz's recent launch [88, 89]) or a shallow reflection of existing access control methodology (as depicted by Facebook in 4.2.3). These models need review and for more intelligence to be included to enhance their usability in a social context.

10.3 Future access control

The state of access control is in flux as the demands and needs of users continues to change and evolve. The demands upon access control are evolving in ways that take it away from the traditional corporate or military approaches. Social access control

is one aspect of this as is a move towards more universal access control. However neither of these on their own appear to be the final solution to access control.

Information is increasingly being generated and classified which leads me to feel that there needs to be a way of generalising access control rules. This thesis provides some solutions to creating generic access control rules that utilise the metadata or semantic information that is increasingly being attached to information. However this is itself not going to be scalable eventually either.

I feel that eventually some for of intelligent agent might end up being the next evolution in access control management. Pattern recognition can form the basis of automatic ruleset generation and may lend itself to either providing rules that are "suggested" to the user or automatically enforcing rules on the fly as access to an item is required. Many of the issues on the web involve the fact that information isn't discoverable because there is no way to obtain access to those resources. With the use of an intelligent agent to mediate the access control, access to information might improve instead of access being denied. Techniques to make access control even smoother remain the future of access control in a way that I feel cannot be imagined today.

Glossary

Access Based Enumeration (ABE) ABE is a feature of Windows Server 2003R2 or higher operating systems that limit the visibility of items that a given user doesn't have the permission to access ordinarily. This is similar to Novell's "file scan" permission however file scan can be removed for an item without changing a users ability to get to that item if they know it exists.. i, 26, 59

Access Control Entry ACE's are entries in a DACL. They usually take the form described in 1.3.4. i, 93

Access Control List See DACL. i, 4, 34, 38, 93

Active Directory Active Directory is Microsoft's LDAP compatible directory services system. Active Directory was introduced with Windows 2000 as the default way of handling distributed identification for users. Active Directory also permits the storage of groups, computers and other objects.. i, 38

Attributes Attributes are flags or name value pairs set on an object typically within a file system. There can be confusion between the user of this and permissions where the attribute flag controls a particular permission. For example the ext2 extended attribute 'append only' could be considered permissive in that with the flag being set it prevents overwriting or truncating the file however the 'sync' attribute controls the kernel behaviour not the permission active for the file. Typically if a file system object has an attribute attached it applies to all users of the file equally regardless of individual permissions or operating system behaviours (e.g. an ext2 immutable file cannot be altered by the root user however the root user may remove the immutable bit and then edit the file). i

Capability Capabilities are a feature of Linux equivalent to Windows privileges. See also: Privilege. i

Container A container is a special object that can contain other objects. Containers can have permissions assigned to them individually and these may or may not be inherited by the child objects of the container. In file systems, containers are typically known as directories or folders.. i

Discretionary Access Control (DAC) DAC is a form of access control that is typically applied to specific objects or subjects. i, 12, 93

Discretionary Access Control Lists (DACL) Discretionary Access Control Lists are used in a DAC environment to enumerate the subjects that can complete actions on an object. DACL's are typically comprised of Access Control Entrys (ACEs). Also known as ACL. i, 34, 93

DN A distinguished name is a string which uniquely defines an entry in an LDAP system. DN's are constructed in a hierarchical manner similar to DNS. i, 38

Down-level Name The down-level name is the username format best known as "DOMAIN
username" which was used in Windows Directory Services prior to Active Directory in Windows 2000. [30]. i, 38

Extended Attributes Extended Attributes have a dual meaning: primarily extended attributes are name value pairs that are stored by the file system however extended attributes can refer to attributes added that aren't standard for those file systems types although built into the file system (for example ext2's extended attributes). Generic extended attributes should be accessible via a common API (get_xattr and set_xattr) while file system specific extended attributes (such as ext2's immutable attribute) require special commands and API's to alter their value.. i

FAT File Allocation Table is a simple file system developed by Microsoft and used initially in MSDOS. i, 2, 59

FUSE File system in User Space is a system to permit file system development outside of the kernel and to expand the availability of languages and libraries that can be used to build a file system.. i, 77

HFS HFS was the successor to MFS for Macintosh computers. HFS introduced a hierarchical approach. HFS is also known as "Mac OS Standard". i, 2

HFS+ HFS+ is an improved version of Apple's Hierarchical File System. HFS+ is referred to as "Mac OS Extended" in comparison to HFS which is referred to as "Mac OS Standard". i, 2, 59

LDAP Lightweight Directory Access Protocol is a standard for accessing directories of information. LDAP is a protocol used to interface with popular user and group directories like Microsoft's Active Directory, Apple's OpenDirectory, Novell's eDirectory and is implemented by the open source "OpenLDAP" project.. i, 38

Mandatory Access Control (MAC) MAC is a form of access control that usually encompass rules that apply to all objects or subjects in the system. MAC rules usually override DAC rules.. i, 12, 34

MDFS MDFS is the underlying system developed at USQ as a research metadata filesystem. Metadata filesystems distinguish themselves from traditional filesystems by providing more advanced metadata interactions and behaviours akin to object orientated system or conventional relational databases. i

MFS MFS was one of the first file systems developed by Apple for the Macintosh computers. MFS was a limited file system with no inherent permissions. i, 2

MSDOS Microsoft Disk Operating System was a single user operating system developed by Microsoft and delivered on IBM PCs. DOS for a period was the dominant operating system for microcomputers. i, 2, 94

MSSQL Microsoft SQL Server is Microsoft's proprietary relational database server. i

MULTICS Multiplexed Information and Computer Service was an early time sharing operating system. MULTICS pioneered many advanced concepts particularly around security.. i, 3

MySQL MySQL is an open source relational database system popular for running web applications. MySQL is used by organisations such as Google to provide their data backends.. i

NFS Network File System developed by Sun Microsystems (now owned by Oracle) is an early UNIX based systems (particularly Solaris). NFS has had many iterations the latest being version 4 which introduced NTFS style ACLs. i, 4

NTFS New Technology File System. Microsoft's preferred file system for Windows NT or later (now Windows Server) and Windows XP or later. NTFS had a significant security focus with the ability to store multiple discretionary ACL rules for files and folders. i, 4, 59

NWFS Netware File System. i, 59

Objects An object is an item that can have permissions applied to it. In file systems an object is typically represented by a file.. i

Oracle Oracle is a software company most known for its proprietary relational database server. Typically when referring to Oracle in a database context, the reference is not to its company but to the relational database management system that the company sells.. i

Permission A permission is the ability to enact, or the inability to enact, an action associated with a capability.. i

Permissive Attribute A permissive attribute is an attribute of a file system that also defines a permission. See attributes for example of permissive and non-permissive attributes. i

POSIX Portable Operating System Interface is a family of standards defined by the IEEE that lay out software standards for operating systems. POSIX is a popular standard in the UNIX and UNIX derivative world with Mac OS X being fully POSIX compliant and Linux regarded as mostly compliant. POSIX is referred to here as short hand for "POSIX.1". i

POSIX ACL POSIX ACL is a draft standard (POSIX.1e) for implementing ACL in POSIX based file systems and operating systems.. i

PostgreSQL PostgreSQL is an open source relational database system that is similar to MySQL but typically touted as having more features. PostgreSQL is noted for being more compatible with Oracle and providing features such as views and stored procedures long before they were available in MySQL.. i

Privilege A privilege is an ability to achieve a particular action which may or may not violate the individual permissions of a file. Privileges may permit control of abilities that aren't ordinarily controlled by the particular file system.. i, 93

Role Based Access Control RBAC is covered in 2.3 as an example of a possible permissions model. Role based access control is the concept of assigning actions to a role and then users are assigned roles. Then users are assigned to various roles which determines their positions. Role based access control can be considered similar to using positions in an organisation. i

Relation Based Access Control RelBAC provides a model of access control by utilising relationships between subjects and objects. RelBAC is covered in 2.5 and 3.6.1. i, 13

SQL Structured Query Language is a programming language based on relational algebra for querying databases. i, 7

Super User A super user is a user who typically inherently has the ability to complete all actions within the system regardless of the actual permissions on the objects though typically a super user cannot overrule attributes (though they can usually alter the attributes to avoid the security enforced by them). i

UPN A user principal name is the preferred login name for Active Directory. The UPN utilises a similar format to an email and is used in the form "username@usq.edu.au" where the username is located at the site name or domain of the particular instance. It is common for the organisation's DNS name (or a subdomain) to be used as the Active Directory site name. The UPN is an instance of a Kerberos principal. The Kerberos protocol is implemented and supported by Active Directory. i, 38

WinFS Windows Future Storage was the planned storage back end for Windows Longhorn (later released as Windows Vista). WinFS was a metadata based file system using a relational data store backend. WinFS as a project was cancelled in 2006.. i, 4

Bibliography

[1] About controlling access to sites and site content [online]. Available from: http://office.microsoft.com/en-us/sharepointtechnology/HA101001441033.aspx [cited 2010-04-12].

[2] About managing sharepoint groups and users [online]. Available from: http://office.microsoft.com/en-us/help/HA100215791033.aspx [cited 2010-04-12].

[3] Conflicts between privileges and permissions [online]. Available from: http://technet.microsoft.com/en-us/library/cc962012.aspx [cited 2010-03-27].

[4] Fileacl [online]. Available from: http://www.gbordier.com/gbtools/fileacl.asp [cited 2010-03-27].

[5] Filesystem in userspace [online]. Available from: http://fuse.sourceforge.net/ [cited 2010-04-18].

[6] Guide to sharepoint server features [online]. Available from: http://msdn.microsoft.com/en-us/library/ms561082.aspx [cited 2010-04-12].

[7] How permissions are calculated [online]. Available from: http://docs.moodle.org/en/How_permissions_are_calculated [cited 2009-06-07].

[8] Microsoft windows xp - fsutil: hardlink [online]. Available from: http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/fsutil_hardlink.mspx?mfr=true [cited 2009-06-27].

[9] Ms security configuration manager for windows nt 4 [online]. Available from: http://technet.microsoft.com/en-us/library/cc723490.aspx [cited 2010-03-27].

[10] Novell doc: Oes 1 - directory and file attributes for nss volumes or netware traditional volumes [online]. Available from: http://www.novell.com/documentation/oes/stor_filesys/?page=/documentation/oes/stor_filesys/data/bs3fkbm.html [cited 2009-06-28].

[11] Novell doc: Oes 1 - file-system trustee rights [online]. Available from: http://www.novell.com/documentation/oes/stor_filesys/?page=/documentation/oes/stor_filesys/data/bt4ampu.html [cited 2009-06-28].

[12] Permission levels and permissions [online]. Available from: http://office.microsoft.com/en-us/sharepointtechnology/HA101001491033.aspx [cited 2010-04-12].

[13] Roles and capabilities [online]. Available from: http://docs.moodle.org/en/Roles_and_capabilities [cited 2009-06-07].

[14] Rule set based access control [online]. Available from: `http://www.rsbac.org/` [cited 2010-04-02].

[15] Understanding container access inheritance flags in windows 2000 [online]. Available from: `http://support.microsoft.com/kb/220167` [cited 2009-06-28].

[16] Mac os x manual page for acl(3) [online]. December 2002. Available from: `http://developer.apple.com/documentation/Darwin/Reference/Manpages/man3/acl.3.html` [cited 2009-07-02].

[17] How ntfs works: Local file systems [online]. March 2003. Available from: `http://technet.microsoft.com/en-us/library/cc781134(WS.10).aspx` [cited 2009-06-27].

[18] Download details: Fileacl [online]. March 2004. Available from: `http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=723f64ea-34f0-4e6d-9a72-004d35de4e64` [cited 2009-06-27].

[19] Mac os x manual page for chmod(1) [online]. July 2004. Available from: `http://developer.apple.com/mac/library/DOCUMENTATION/Darwin/Reference/ManPages/man1/chmod.1.html` [cited 2009-07-02].

[20] Exploring tiger server [online]. April 2005. Available from: `http://developer.apple.com/server/tigerserver.html` [cited 2009-07-02].

[21] Zfs acls [online]. November 2005. Available from: `http://blogs.sun.com/marks/entry/zfs_acls` [cited 2009-07-02].

[22] Linux shadow password howto [online]. April 2006. Available from: `http://tldp.org/HOWTO/Shadow-Password-HOWTO.html` [cited 2009-06-07].

[23] Php generic access control lists [online]. September 2006. Available from: `http://phpgacl.sourceforge.net/` [cited 21-May-2009].

[24] Welcome to subsystem for unix-based applications [online]. March 2006. Available from: `http://technet2.microsoft.com/WindowsServer/en/Library/695ac415-d314-45df-b464-4c80ddc2b3bc1033.mspx` [cited 2009-08-07].

[25] hard links to directories [lwn.net] [online]. September 2007. Available from: `http://lwn.net/Articles/249607/` [cited 2009-06-27].

[26] Issue 5 - xar - archiving acls on mac os x 10.4 doesn't work - google code [online]. February 2007. Available from: `http://code.google.com/p/xar/issues/detail?id=5` [cited 2009-07-02].

[27] Leopard technology series for developers: Os foundations [online]. October 2007. Available from: `http://developer.apple.com/leopard/overview/osfoundations.html` [cited 2009-08-07].

[28] Mediawiki [online]. January 2007. Available from: `http://www.mediawiki.org/wiki/MediaWiki` [cited 2010-08-09].

[29] Overview of fat, hpfs and ntfs file systems [online]. May 2007. Available from: `http://support.microsoft.com/kb/q100108/` [cited 2009-06-07].

[30] Users can log on using user name or user principal name [online]. March 2007. Available from: `http://support.microsoft.com/kb/243280` [cited 2010-05-05].

[31] You cannot configure ntfs permissions to hide files or folders from unauthorized users [online]. March 2007. Available from: http://support.microsoft.com/kb/303758 [cited 2009-06-28].

[32] ipod: How to determine your ipod's disk format [online]. June 2008. Available from: http://support.apple.com/kb/HT1335 [cited 2009-07-02].

[33] link (the open group base specifications issue 7) [online]. 2008. Available from: http://www.opengroup.org/onlinepubs/9699919799/functions/link.html [cited 2009-06-27].

[34] 5.4.1: Privileges provided by mysql [online]. May 2009. Available from: http://dev.mysql.com/doc/refman/5.4/en/privileges-provided.html [cited 10-May-2009].

[35] Access-checking [online]. 2009. Available from: http://technet.microsoft.com/en-us/library/cc962006.aspx [cited 2009-06-28].

[36] Access masks [online]. May 2009. Available from: http://msdn.microsoft.com/en-us/library/ms790780.aspx [cited 1-June-2009].

[37] Ace inheritance rules (windows) [online]. June 2009. Available from: http://msdn.microsoft.com/en-us/library/aa374924(VS.85).aspx [cited 2009-06-28].

[38] D (windows) - security glossary [online]. June 2009. Available from: http://msdn.microsoft.com/en-us/library/ms721573(VS.85).aspx#_security_discretionary_access_control_list_gly [cited 2009-06-28].

[39] Order of aces in a dacl [online]. 2009. Available from: http://technet.microsoft.com/en-us/library/cc961994.aspx [cited 2009-06-28].

[40] Order of aces in a dacl (windows) [online]. June 2009. Available from: http://msdn.microsoft.com/en-us/library/aa379298(VS.85).aspx [cited 2009-06-28].

[41] Privileges [online]. November 2009. Available from: http://msdn.microsoft.com/en-us/library/ff551855(VS.85).aspx [cited 2010-04-04].

[42] Security features for file systems [online]. May 2009. Available from: http://msdn.microsoft.com/en-us/library/ms791051.aspx [cited 01-June-2009].

[43] Access control entries [online]. 2010. Available from: http://technet.microsoft.com/en-us/library/cc961995.aspx [cited 2010-04-05].

[44] ACE_HEADER Structure [online]. February 2010. Available from: http://msdn.microsoft.com/en-us/library/aa374919(VS.85).aspx [cited 2010-04-05].

[45] Content management system - docforge programming wiki [online]. June 2010. Available from: http://docforge.com/wiki/Content_management_system [cited 2010-08-08].

[46] drupal.org — community plumbing [online]. August 2010. Available from: http://drupal.org [cited 2010-08-08].

[47] Dublin core metadata initiative [online]. August 2010. Available from: http://dublincore.org [cited 2010-08-08].

[48] Joomla! [online]. August 2010. Available from: `http://www.joomla.org` [cited 2010-08-08].

[49] Openid - features - crowd [online]. August 2010. Available from: `http://www.atlassian.com/software/crowd/features/openid.jsp` [cited 2010-08-13].

[50] Privilege constants [online]. 02 2010. Available from: `http://msdn.microsoft.com/en-us/library/bb530716(VS.85).aspx` [cited 2010-03-28].

[51] setuid [online]. March 2010. Available from: `http://en.wikipedia.org/wiki/Setuid` [cited 2010-04-05].

[52] Sticky bit [online]. March 2010. Available from: `http://en.wikipedia.org/wiki/Sticky_bit` [cited 2010-04-05].

[53] Web content management system [online]. August 2010. Available from: `http://en.wikipedia.org/wiki/Web_content_management_system` [cited 2010-08-08].

[54] What is a cms? [online]. July 2010. Available from: `http://www.cmscritic.com/what-is-a-cms/` [cited 2010-08-08].

[55] Wordpress ¿ blog tool and publishing platform [online]. August 2010. Available from: `http://wordpress.org` [cited 2010-08-08].

[56] Paul Adams. The real life social network [online]. July 2010. Available from: `http://www.slideshare.net/padday/the-real-life-social-network-v2` [cited 2010-07-18].

[57] Apple. *sticky*, 4th berkeley distribution edition, June 1993. Available from: `http://developer.apple.com/mac/library/DOCUMENTATION/Darwin/Reference/ManPages/man8/sticky.8.html` [cited 2010-04-05].

[58] Apple. *link(2)*, October 2008. Available from: `http://developer.apple.com/mac/library/documentation/Darwin/Reference/ManPages/man2/link.2.html` [cited 2010-04-05].

[59] N Appliance. Merging nt and unix filesystem permissions. *usenix.org*. Available from: `http://www.usenix.org/publications/library/proceedings/lisa97/failsafe/usenix-nt98/full_papers/hitz/hitz_html/`.

[60] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986.

[61] M Berger. Silenus-a federated service-oriented approach to distributed file systems. *etd.lib.ttu.edu*, Jan 2006. Available from: `http://etd.lib.ttu.edu/theses/available/etd-11062006-142552/`.

[62] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley, Boston, MA, USA, December 2002.

[63] A Bohra, S Smaldone, and L Iftode. Frac: Implementing role-based access control for network file systems. *Sixth IEEE International Symposium on Network Computing and ...*, Jan 2007. Available from: `http://www.cs.rutgers.edu/~iftode/FRAC07.pdf`.

[64] Jonathan Corbet. A bid to resurrect linux capabilities [online]. 2006. Available from: `http://lwn.net/Articles/199004/` [cited 2010-03-28].

[65] Ramez Elmasri and Sham Navathe. Fundamentals of database systems. page 1030, Jan 2004. Available from: http://books.google.com/books?id=zlfiPwAACAAJ&printsec=frontcover.

[66] Paul Fenwick. Pjf's pages - journal - table of contents - facebook [online]. 2010. Available from: http://pjf.id.au/blog/toc.html?tag=facebook [cited 2010-05-05].

[67] F Giunchiglia, Rui Zhang, and B Crispo. Relbac: Relation based access control. *Semantics, Knowledge and Grid, 2008. SKG '08. Fourth International Conference on*, pages 3 – 11, Nov 2008. Very interesting paper. Available from: http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4725889&isnumber=4725879&punumber=4725878&k2dockey=4725889@ieeecnfs, doi:10.1109/SKG.2008.76.

[68] Apple Insider. Road to mac os x leopard: Time machine [page 3] [online]. October 2007. Available from: http://www.appleinsider.com/articles/07/10/12/road_to_mac_os_x_leopard_time_machine.html&page=3 [cited 2009-06-27].

[69] Samyak Jain. Working with filesystems using nfsv4 acls [online]. June 2009. Available from: http://www.ibm.com/developerworks/aix/library/au-filesys_NFSv4ACL/index.html [cited 2009-07-02].

[70] J. Kunze and T. Baker. The Dublin Core Metadata Element Set. RFC 5013 (Informational), August 2007. Available from: http://www.ietf.org/rfc/rfc5013.txt.

[71] L LaPadula. Rule-set modeling of trusted computer system. Jan 1995. Available from: http://madchat.awired.net/reseau/defense/l693ibfb.pdf.

[72] David MacKenzie and Jim Meyering. *chmod(1)*. Free Software Foundation, 2006. Available from: http://linux.die.net/man/1/chmod [cited 2010-04-05].

[73] Matt McKeon. The Evolution of Privacy on Facebook [online]. April 2010. Available from: http://mattmckeon.com/facebook-privacy/ [cited 2010-05-09].

[74] Greg Miller. Exploring leopard with dtrace. *MacTech*, 23(11), 2007. Available from: http://www.mactech.com/articles/mactech/Vol.23/23.11/ExploringLeopardwithDTrace/index.html [cited 2009-06-27].

[75] Sam Moffatt. There and back again: a history of access control systems. In *MSC Malaysia Open Source Conference*. MSC Malaysia, MSC Malaysia, July 2010. Available from: http://eprints.usq.edu.au/8331/.

[76] Sam Moffatt and Paul Graham. Permissions in viaweb. email, 2009 August.

[77] Tony Northrup. *Introducing Microsoft Windows 2000 Server*. Microsoft Press, 1999. Available from: http://technet.microsoft.com/en-au/library/bb742424.aspx.

[78] G Nuremberg. Posix access control lists on linux. *usenix.org*. Available from: https://www.usenix.org/publications/library/proceedings/usenix03/tech/freenix03/full_papers/gruenbacher/gruenbacher.pdf.

[79] I Olson and M Abrams. Computer access control policy choices. *Computers & Security*, 9(8):699–714, Dec 1990. Available from: http://linkinghub.elsevier.com/retrieve/pii/0167404890901138, doi:10.1016/0167-4048(90)90113-8.

[80] OpenID Foundation. Openid foundation website [online]. July 2010. Available from: http://openid.net/ [cited 2010-07-30].

[81] Kurt Opsahl. Facebook's eroding privacy policy: A timeline [online]. April 2010. Available from: http://www.eff.org/deeplinks/2010/04/facebook-timeline [cited 2010-05-05].

[82] Wayne Pollock. Unix file and directory permissions and modes [online]. 2009. Available from: http://content.hccfl.edu/pollock/AUnix1/FilePermissions.htm [cited 2010-04-05].

[83] Brent Rector. *Introducing Longhorn for Developers*. Microsoft Press, October 2003. Available from: http://msdn.microsoft.com/en-us/library/aa479870.aspx [cited 2009-08-11].

[84] R Reichel. Inside windows nt security. *Windows/DOS Developer's Journal*, Jan 1993. Available from: http://portal.acm.org/citation.cfm?id=159896.159898.

[85] Mark Russinovich. Windows nt security, part 1 [online]. October 2002. Available from: http://www.windowsecurity.com/whitepapers/Windows_NT_Security_Part_1.html [cited 2009-08-07].

[86] Mark Russinovich. The bypass traverse checking (or is it the change notify?) privilege [online]. October 2005. Available from: http://blogs.technet.com/markrussinovich/archive/2005/10/19/the-bypass-traverse-checking-or-is-it-the-change-notify-privilege.aspx [cited 2010-04-02].

[87] R Russon and Y Fledel. Ntfs documentation. *Linux-NTFS.[Online]*, Jan 2005. Available from: http://ronis.liis.lv/~rihardr/stud/hd/fs/ntfsdoc.pdf.

[88] Nick Saint. Google responds to blogger's outrage with product tweaks. *Business Insider*, February 2010. Available from: http://www.businessinsider.com/google-adding-two-privacy-features-in-response-to-bloggers-outrage-2010-2.

[89] Nick Saint. Outraged blogger is automatically being followed by her abusive ex-husband on google buzz read more: http://www.businessinsider.com/outraged-blogger-is-automatically-being-followed-by-her-abusive-ex-husband-on-google-buzz-2010-2. *Business Insider*, February 2010. Available from: http://www.businessinsider.com/outraged-blogger-is-automatically-being-followed-by-her-abusive-ex-husband-on-google-buzz-2

[90] V Samar and R Schemers. Unified login with pluggable authentication modules (pam). OSF RFC86, October 1995. Available from: http://www.opengroup.org/rfc/mirror-rfc/rfc86.0.txt [cited 2010-03-28].

[91] R Sandhu, E Coyne, H Feinstein, and C Youman. Role-based access control models. *Computer*, Jan 1996. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=485845.

[92] Michael Shaffer. Filesystem security - ext2 extended attributes [online]. November 2000. Available from: http://www.securityfocus.com/infocus/1407 [cited 2009-06-07].

[93] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) version 4 Protocol. RFC 3530 (Proposed Standard), April 2003. Available from: http://www.ietf.org/rfc/rfc3530.txt.

[94] Avi Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, seventh edition edition, 2005.

[95] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). RFC 2251 (Proposed Standard), December 1997. Obsoleted by RFCs 4510, 4511, 4513, 4512, updated by RFCs 3377, 3771. Available from: http://www.ietf.org/rfc/rfc2251.txt.