University of Sussex

Secure File Sharing

Rakan Alsowail

Submitted for the degree of Doctor of Philosophy

University of Sussex

February 2016

UNIVERSITY OF SUSSEX

RAKAN ALSOWAIL, Doctor of Philosophy

Secure File Sharing

Summary

File sharing has become an indispensable part of our daily lives. The shared files might be sensitive, thus, their confidentially, integrity and availability should be protected. Such protection might be against *external threats* that are initiated by unauthorised users or *insider threats* that are initiated by authorised users. Our main interest in this thesis is with insider threats. Protecting shared files against insiders is a challenging problem. Insiders enjoy various characteristics such as being trusted and authorised, in addition to being inside the network perimeter and having knowledge of information systems. This makes it difficult to prevent or detect policy violation for these users. The goal of this thesis is to protect shared files from the perspective of insider security with language-based techniques.

In the first part of the thesis, we define what we mean by an *insider* and the *insider problem* precisely, and propose an approach to classify the insider problem into different categories. We then define and focus on one category that is related to file sharing. Namely, protecting the confidentiality and integrity of the shared files against accidental misuse by insiders. Furthermore, we classify the activity of file sharing into different categories that describe all possible ways of performing the activity of file sharing. These categories represent policies that describe how files should be propagated and accessed by insiders. We show that enforcing these policies can protect the files against accidental misuse by insiders while allowing the activity of sharing to be performed as desired. Thus our interest can be summarised as keeping honest users safe.

In the second part of the thesis, we develop a security type system that statically enforces information flow and access control policies in a file system. Files are associated with security types that represent security policies, and programs are sets of operations to be performed on files such as read, copy, move, etc. A type checker, therefore, will statically check each operation to be performed on a file and determine whether the operation satisfies the policy of the file. We prove that our type system is sound and develop a type reconstruction algorithm and prove its soundness and completeness. The type system we developed in this thesis protects the files against accidental misuse by insiders.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor Dr. Ian Mackie, for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better supervisor and mentor for my Ph.D study.

Besides my supervisor, I would like to thank Prof. Ian Wakeman, Dr. Dan Chalmers, Prof. Maribel Fernandez, Dr. Abubakar Hassan, and Dr. Nikolaos Siafakas, for their insightful comments and encouragement, but also for the hard question which urged me to widen my research from various perspectives.

My sincere thanks also goes to my friends from Sussex, for their support, patience and words of advice. In particular I am grateful to Dr. Renan Krishna and Dr. Shinya Sato. Thanks also to all my friends for being the lovely and supportive people they are.

I would like to thank my deceased father, who I am sure would have been proud of my work, and my mother for their great role in my life and their numerous sacrifices for me and for giving me the courage and strength to face obstacles that come my way. I would also like to thank my brothers and sisters for supporting me spiritually throughout writing this thesis and my life in general.

Lastly but most importantly, A big thank you to my daughters Hala and Leen, and my wife Rabab for their love and for standing with me during the hard times. I would also like to thank them for tolerating my long working hours in the lab and time spend away from them. I would specially like to thank my wife for her encouragement over the years and for not letting me give up on my dream, without her love, support, and belief in me, I would have never completed my Ph.D studies.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter gives an overview of the thesis, explains its rationale and describes the contributions to knowledge that it makes. The overall structure of the thesis is also presented.

1.1 Introduction

With the advent of Web 2.0, Internet users have become more active than ever before. They have changed from being passive users that consume content to active users that produce on-line content. Web 2.0 allows Internet users' to be producers and consumers of on-line content at the same time. Examples of internet users content are blogs, wikis, documents (such as Google Docs), multimedia (i.e. pictures, videos, music), and personal bibliographical information. One of the prominent characteristics of Web 2.0 is that users are able to generate and share content on the Web without special technical skills [118]. This characteristic has led the majority of Internet users to generate and share their content on-line with one another. According to Mendelsohn and Mckenna [67], 75% of people are somewhat or highly likely to share content they like on-line with friends, co-workers or family, and 49% share content on-line at least once a week. The authors point out that more than 30 million pieces of content (web links, news stories, blog posts, notes, photo albums, etc.) are shared each month on Facebook alone. In this thesis we will use the term *file sharing* instead of content sharing where a file can be a picture, an audio, a video, or text, etc.

Files can be classified into commercial and non-commercial, where noncommercial files can be further classified into confidential and non-confidential (See Figure 1.1). Commercial and non-commercial but confidential files are sensitive which means that they need to be protected from potential attacks or misuses. Such attacks might lead to unauthorised disclosure (Confidentiality attacks), unauthorised modification (Integrity attacks), or unauthorised withholding (Availability attacks). These different types of attacks can be performed on files while they are being transferred, stored, or used by either authorised or unauthorised users. As a result, protections for these types of attacks stem from three distinct fields of security which are: Communication security which is concerned with preventing different types of attacks on data transmitted over a network; Perimeter security which is concerned with preventing attacks on data stored inside a trusted internal network; and Insider security which is concerned with preventing attacks on data by those who have been authorised with access.

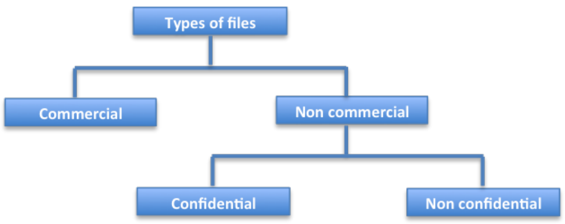

Figure 1.1: Types of content

This chapter discusses issues arising with protecting commercial and confidential files from the perspective of insider security. In particular, issues arising with protecting the shared files against attacks that are performed by users who are authorised to access the files.

1.2 Level of Trust and Protection

The goal of protecting commercial files is different from that of protecting confidential files. In commercial files, the goal is to prevent access to the files by users who do not pay for access, while the goal of protecting confidential files is to prevent access to the files by users who are not authorised by the files' owners, whether they pay to access the files or not. Although the two types of files might require the same type of protections (i.e. confidentiality, integrity, and availability protections), the strength level of the required protection is different. This is due to the fact that users who are authorised to access commercial files have different trust levels from those who are authorised to access confidential files.

The required strength level of the protection mechanisms for commercial and confidential files is determined by the level of trust that is maintained by those with whom the file is shared. That is, low trust implies that a stronger level of protection mechanism is needed, whereas high trust implies a weaker level of protection mechanism. This is because it is perceived that trusted parties will not violate the file policy, hence, protection is not required, whereas untrusted parties might violate the file policy, hence, a strong protection is required. Those who are neither trusted nor untrusted might require a moderate level of protection to avoid unnecessary costs incurred by excessive protection mechanisms.

It should be noted that while low trust implies the need for a stronger protection mechanism, the stronger the protection mechanism is, the more cost is incurred. By cost we do not only mean the monetary cost, but also the cost of usability and privacy as implementing a strong protection mechanism makes the usage inconvenient [120], and might require the collection of information about the usage such as in Digital Rights Management systems (DRM) [58] and Intrusion Detection Systems (IDS) [56]. Figure 1.2 illustrates that the less trust there is, the stronger the security mechanism is required which entails more cost, whereas the more trust there is, the weaker the security mechanism is required which entails less cost.

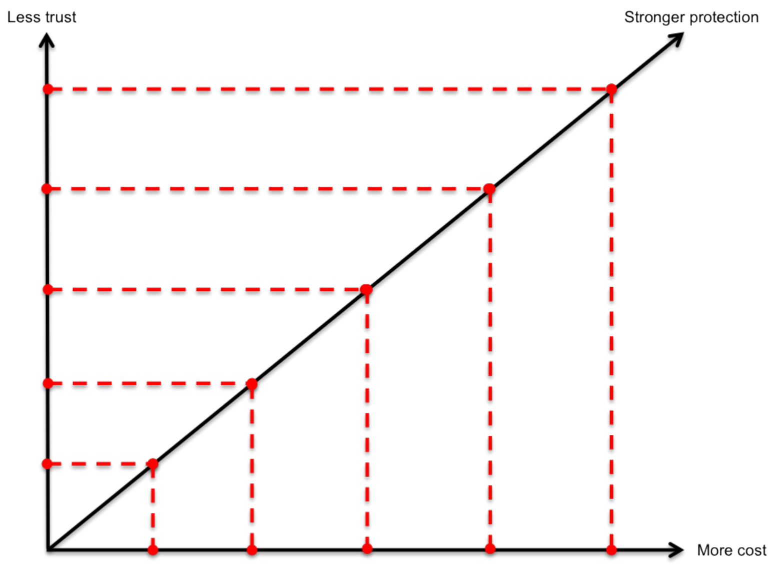

Figure 1.2: Trust vs. Protection vs. cost

Deciding whether to implement a particular level of protection mechanism is determined by evaluating the cost of the level of the protection against the value of the content which needs to be protected. Such evaluation helps in finding out whether the cost is worth protecting the content or not.

Since the commercial files are shared with users based on payment, the authorised users of commercial files are considered untrustworthy and usually referred to in the literature as adversaries. Therefore, the strongest possible level of protection is required. On the other hand, confidential files are shared with users based on a certain level of trust. However, levels of trust might range from trusted to entirely untrusted and which entails disparate levels of protections that range from no protection to the strongest level of protection. Figure 1.3 illustrates the various levels of trust and protection that might be required to protect commercial and confidential files.

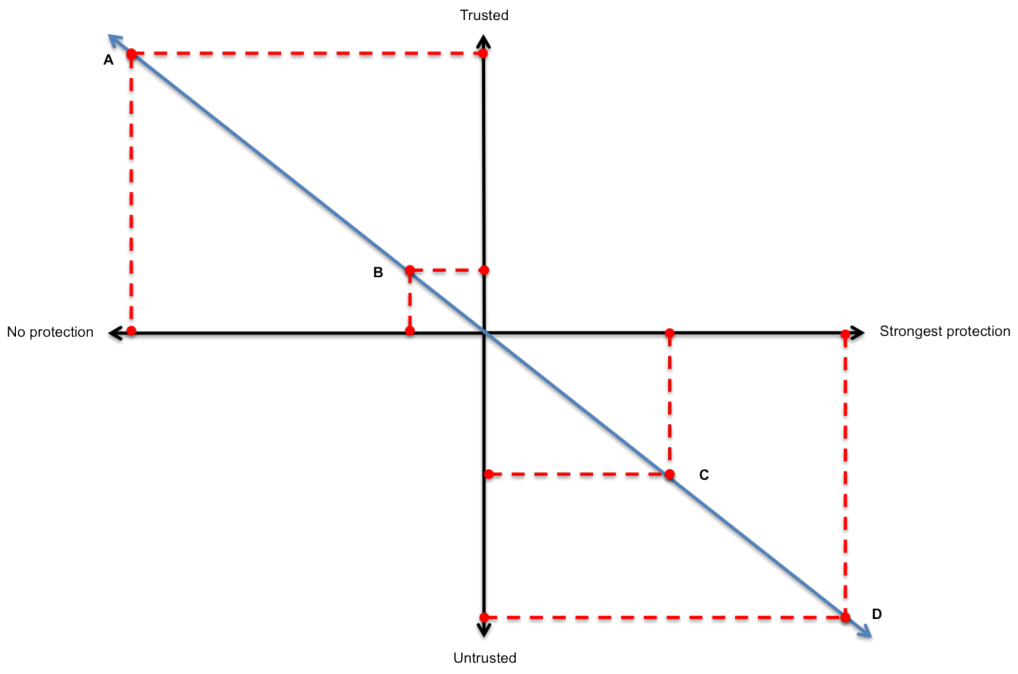

Figure 1.3: The optimal level of protection

As shown in Figure 1.3, the blue arrow across the area *Trusted - No protection and Untrusted - Strongest protection* is used to identify the different optimal levels of protection based on different degrees of trust. The area between *Untrusted - No protection* and *Trusted - Strongest protection* is usually perceived as to describe inappropriate levels of protection, and therefore is omitted. For instance, providing the strongest level of protection against trusted individuals results in unnecessary costs, while providing no protection against untrustworthy individuals is risky.

The four red points on the blue arrow indicate four levels of protection that are required by the four levels of trust. Level A illustrates the case where individuals are trusted and therefore no protection is required. On the other hand, level D illustrates the case where individuals are untrusted and therefore the strongest level of protection is required. Between level A and D, various levels of trust might exist, each requiring a particular level of protection (e.g. level B and C). In the next section, two issues of trust and protection are illustrated.

1.3 Problem Statement

This thesis is concerned with the problem of protecting the shared files against authorised users in a Unix-like file system. Users in the file system manipulate files through issuing various commands such as `mv`, `cat`, and `cp`. In the file system, sharing is performed through commands that cause information to flow between files which are accessed by different users. For example, copying a file which can only be accessed by *Alice* into another file which can be accessed by *Bob* and *Carol* is considered sharing. This thesis aims to apply a language-based technique, particularly a type system, to protect the shared files against commands issued to manipulate them by authorised users.

Authorised users can be classified into trusted and untrusted users. In this section, we present two issues of trust and protection. The first is concerned with protecting the shared files against untrusted authorised users. Whereas the second is concerned with protecting the shared files against trusted authorised users. The focus of this thesis is on protecting the shared files against trusted authorised users. However, we first present the issue of protection against untrusted authorised users to highlight and show the significance of the protection against trusted authorised users. In Section 1.4 we discuss the approach taken in this thesis to solve the above problem in detail.

1.3.1 Untrusted individuals - Strongest protection (D)

At one end of the blue arrow in Figure 1.3 are untrusted individuals where the strongest level of protection is required. This is usually the case with commercial files and some cases of confidential files (e.g. military and intelligence settings).

Providing a protection mechanism that completely prevents file misuse by entirely untrusted individuals is a dilemma. Research and history have shown that such a mechanism does not exist and is an impossibility, as there is no system which is 100% secure against all deliberate attacks or misuse [87, 103, 46, 94]. A brief look at the approach taken to protect commercial content, justifies this principle. Commercial content is protected by the use of DRM systems that dictate how the content must and must not be used by each individual. Examples of DRM systems are Windows Media DRM [136], Apple FairPlay [32], Adobe PDF DRM [31], and SecuRom [105], that provide protection for different types of files such as video, music, documents, or games.

Although these systems are in place to protect the commercial files and they are supposed to prevent all possible misuses, the files can still be obtained illegally in unprotected form. This cause stems from two reasons as follows.

Systems vulnerabilities:

Each system has its own vulnerabilities and there is no system without vulnerabilities [103, 5]. A system's vulnerabilities can be easy or hard to find and, of course, vulnerabilities of DRM systems are hard to find as much effort is dedicated to produce a highly secure system. It is hard for an average user to find such vulnerabilities in DRM systems, however, a more sophisticated and determined attacker who has the time and knowledge might spend days, weeks, or even months to ultimately circumvent the system. Once the system is circumvented by that sophisticated attacker, the content becomes unprotected, and can then be provided to average users freely without protection by various means such as peer-to-peer networks [40]. Worse, the attacker might develop an automated tool to launch his attack and distribute it to average users so that they can circumvent their systems without the need to learn any sophisticated technical skills [46]. In fact, DRM systems rely on the concept of security-through-obscurity which is a principle in security engineering that means the security of a system is provided by the secrecy of its design or implementation. Therefore, once the sophisticated attacker has figured out the inner design of the system, he will be able to circumvent it [114, 69, 115, 58]. Studies have shown that all DRM systems can be circumvented [94, 12]. Various successful attacks on DRM systems are discussed in detail in [46, 69]

The same thing is applied to other systems that are used to protect confidential files from misuse by entirely untrusted individuals. For instance, the recent leak of more than 200,000 classified documents by the former NSA contractor Edward Snowden [133], is evidence that regardless of the protection system in place, a determined attacker will still be able to circumvent it.

The analog hole:

One of the easiest ways to circumvent any protection system used to protect commercial or confidential files is by exploiting the analog hole [138]. The analog hole is the inevitable vulnerability in any file protection system that makes the protection of content from untrusted individuals unfeasible [115, 12]. Any digital file must eventually be converted to human-perceptible form, which is known as the analog form, in order to be consumed by the users. Once the digital file is converted to analog form, it will be in an unprotected form, and thus, susceptible to unauthorised uses [40, 138].

For instance, protected music played in a computer is converted to sound waves whereby a user can record the audio coming out of the computer speaker without any restriction. Also, a protected video, text and photo displayed on a computer screen are converted to light patterns and can be video recorded, or captured by a digital camera without any restriction. These are examples of exploiting the analog hole and include memorising content as the human brain is able to memorise information for sometime. Such exploitation of the analog form of protected digital files is hard or even impossible to prevent and usually referred in the literature to as the analog hole problem [115, 138].

For these two reasons, any protection mechanism that protects files from entirely untrusted individuals will ultimately fail. Any commercial and confidential file has a time value. The time value of a file might be days, weeks or months, after which the file will lose its value. For instance, a commercial file which is sold now at a particular price will decrease in value over time until it loses its value and becomes free. Also, a confidential file which must be kept secret now, at later time might need to be publicly published and thus loses its value and is no longer secret. Therefore, protection mechanisms might be useful for protecting a file that has a short time value. This is because the time that is needed to circumvent the protection mechanism might exceed the time value of the file. In other words, by the time an attacker manages to circumvent the protection mechanism, the file will have lost its value and little or no loss will be occurred if the file is misused. However, by exploiting the analog hole, the protection mechanism can be circumvented immediately. In case of protecting a commercial file, the quality of the content captured in an analog form is usually degraded [12], and therefore, the media and entertainment industry might be tolerant of such exploitation as long as there are people willing to pay for a good quality content even if it is protected.

However, in case of protecting confidential files, degradation in the quality of the content does not matter. What matters is the information in the file regardless of its quality as long as it can be perceived by humans. Additionally, there exist confidential files which have infinite time value, which means that the file will be of high value for the whole of its lifecycle until it is destroyed. In other words, the file must be kept secret forever. Based on the two reasons mentioned previously, protection mechanisms cannot guarantee to protect such files that need to be protected forever. Rather, protection mechanisms can only guarantee to make it harder for an attacker to circumvent the mechanism; perhaps till the file loses its value. Therefore, protection mechanisms might be useful in protecting files that have a short time value rather than content that has long or infinite time value. Protecting confidential files that have long or infinite time value can only be achieved by releasing the files only to trusted individuals who will not violate the content policy deliberately.

1.3.2 Trusted individuals - No protection (A)

At the other end of the blue arrow in Figure 1.3 are trusted individuals where no protection is required. Such individuals are entirely trusted to not violate the content policy, and hence, no protection is usually in place. However, even if individuals are trusted to not violate the content policy deliberately, there is a chance of their violating the policy accidentally. According to a recent survey conducted by Infosecurity Europe and PwC on 1,402 UK companies, 36% of the worst security breaches in the year were caused by inadvertent human error [34]. Also, AngloSec conducted a survey on 197 network, security, and compliance professionals, and found out that the greatest security concern of their respondents was employees accidentally jeopardising security through data leaks or similar errors (40.5%), followed by employees deliberately breaching the security (22.1%)[4].

Furthermore, the Ponemon Institute conducted a survey on 709 IT security practitioners in the United States, and found out that 78% of their respondents have experienced a data breach as a result of negligent or malicious employees or other insiders [54]. Their survey result shows that the root causes of data breach incidents in organisations are loss of laptops or other mobile devices (35%), third party mishaps or flubs (32%), system glitches (29%) mishandling of data at rest and in motion (27% and 23% respectively), malicious employees or other insiders (22%), and external cyber attack (8%). Also, they found out that even when misuses are unintentionally made by employees, most of these misuses are discovered accidentally and rarely self-reported by the employees themselves.

Accidental breach in security can also occur during the activity of file sharing. For example, a confidential file might be shared accidentally with unintended recipients, or overwritten accidentally by irrelevant content. While such activity might not be the main reason for the insider problem, it is considered a class of the insider the problem that must be tackled.

Therefore, while confidential files which have long or infinite time value should only be released to entirely trusted individuals to be fully protected, releasing confidential files to entirely trusted individuals without any protection might result in accidental violation of the file policy. There must be an appropriate level of protection that prevents such accidental misuses by those trusted individuals. This appropriate level of protection should prevent all possible violation of content policy that can happen accidentally but ignores those who deliberately circumvent the system; this because we are dealing here with trusted individuals, who are assumed to not violate the content policy deliberately. This thesis focuses on this type of protection.

The broad problem statement that the work reported in this thesis addresses can thus be expressed as follows:

To design a language that allows owners of files to express various polices which define access and usage restrictions on their shared files, and a system that can enforce these polices to prevent accidental misuses of the shared files by trusted recipients.

1.4 Approach

The problem of protecting the shared files from authorised users is considered an instance of the insider threat problem. Therefore, we explore the problem from the perspective of the insider threat. However, the literature on the insider threat problem shows various definitions and understandings of the terms *insider* and *insider threat*. There is no consensus among researchers regarding who is an insider and what are the insider threats. In fact, the insider threat problem is a significant issue, and there is no single definition of the insider and insider threat can encompass the whole problem; though most researchers attempt to provide one. Therefore, we propose an approach to classify the insider threat problem into different categories that can be defined, studied, and solved independently and which later can be combined to solve the problem as a whole. Based on this approach we define a particular category of the insider threat problem that we study and solve throughout this thesis; namely, preventing confidentiality and integrity attacks on files by recipients during the activity of file sharing. However, confidentiality and integrity attacks can be performed in different ways which require different types of protections. Therefore, we investigate the different types of misuse that can be performed by insiders during the activity of file sharing and we characterise the protection required against them. We focus on accidental misuse that affects the confidentiality and integrity of files by trusted insiders. We define our category of the insider problem precisely so as to prevent accidental misuse of confidentiality and integrity of the shared files by trusted recipients during the activity of file sharing.

Although our category of the insider problem and the misuse we need to prevent are precisely defined, the activity of file sharing is still ambiguous. The activity of file sharing can be performed in different ways. Designing a protection mechanism to protect the shared files without taking into account how the activity of file sharing is performed might prevent users from sharing their files as desired. There is a large body of work that investigates the activity of file sharing; however, the majority of it is focused on specific domains and applications. Works that investigate the activity of file sharing generally, miss answers to two fundamental questions that lead to a better understanding of the activity of file sharing; namely, how files can be propagated from owners to recipients and how files can be accessed by the recipients. Therefore, we characterise the activity of file sharing based on how files can be propagated and accessed after their propagation. Based on this characterisation, we define a framework that classifies the activity of file sharing into different categories. Each category specifies how files should be propagated and accessed after their propagation. We show that these categories can be thought of as policies that, if enforced, allow the provision of various types of protection against accidental misuse.

We use a language-based technique to enforce these policies, particularly by the use of a type system; developed to statically enforce information flow and access control requirements of these policies in a file system. As a starting point, we focus on enforcing a particular policy; namely, limiting the number of times a file can be read. Other policies can be enforced similarly, as we discuss in Chapter 6. In the file systems, files are associated with security types that represent the security policies, and programs are sets of operations to be performed on files such as read, copy, move, etc. The type system, therefore, will statically analyse each operation to be performed on a file and determine whether or not the operation satisfies the information flow and access control requirements of the policy associated with the file. Therefore, a type checker which implements the type system will intercept each operation to be performed in the file system and allow only those operations which satisfy the policies of files the operation is performed on. In such a way, the type checker can be thought of as a reference monitor that prevents accidental misuse of the shared files.

1.5 Summary of Contributions

The following list summarises the contributions this research achieves.

- It provides a precise definition of the insider and the insider problem that makes a clear distinction between insiders and outsiders, and between insider threats and external threats, and that encompasses all classes of the insider problem.
- It proposes an approach to classify the insider threat problem into different categories of sub-problems that can be defined, studied and solved independently, and that enables insiders and their threats to be clearly identified in each category.

- It defines a particular class of the insider problem that is related to file sharing and investigates the different types of misuse of the shared files that can be performed by different types of insider, and characterise the protection required against them.
- It characterises the activity of file sharing based on how files can be propagated and accessed after their propagation, and defines a framework based on this characterisation that can classify the activity of file sharing into different categories that can describe all possible ways of how file can be shared, and that can be used to specify policies of files to describe how files should be propagated and accessed after their propagation.
- It develops a language that allows owners of files to express various policies which define access and usage restrictions on their shared files, and a type system that can enforce these policies to prevent accidental misuses of the shared files by trusted recipients.

1.6 Thesis Organisation

The chapters of this thesis are structured in the following way.

Chapter 2: Background In this chapter, we provide the necessary background and related work for topics discussed throughout the thesis. It is divided into three parts. The first part is concerned with file sharing and reviews the history of file sharing and the related work on the methods and people practices of file sharing. The second part is concerned with security in general, and reviews the goals of information security, and related work on communication security, perimeter security and insider security. Our focus will be on insider security since it is relevant to our work. The third part is concerned with language-based security, and reviews related work on access control, information flow control and type systems that enforce information flow control and access control.

Chapter 3: The Insider Threat Problem In this chapter, we address two fundamental questions on the development of a protection mechanism against insider threats. These questions are: ‘what is the insider problem?’ and ‘what is the insider misuse?’. We propose a new approach for classifying the insider threat problem into different categories which can be defined, studied and solved independently and which later can be combined to solve the problem as a whole. Then, we define the insider and the insider problem precisely, and focus on one category that is related to file sharing. We investigate the different kinds of misuse that can be performed by insiders during the activity of file sharing, and characterise the protection required against them. We end the chapter by defining the class of the insider problem that we tackle throughout the thesis; namely the prevention of accidental misuse that affects the confidentiality and integrity of sensitive files during the activity of file sharing.

Chapter 4: Characterising the Activity of File Sharing In this chapter, we characterise the activity of file sharing based on two factors: how files can be propagated from owners to recipients, and how files can be accessed by the recipients after their propagation. Based on the characterisation of the activity of file sharing, we define a framework that classifies the activity of file sharing into different categories. These categories can be thought of as policies that describe how files should be propagated and accessed and that satisfy different sharing scenarios. We show how the framework can be applied to classify the activity of file sharing in an organisation and also to classify existing file sharing methods. We end this chapter by showing how these policies, if enforced, can protect the shared files against the accidental misuse identified in Chapter 3.

Chapter 5: Secure File System In this chapter, we start designing a language that allows owners to specify various policies identified in Chapter 4, and a type system to enforce these policies. As a starting point, this chapter focuses on enforcing a particular policy; namely, limiting the number of times a file can be read. Other policies can be similarly enforced, as we discuss in Chapter 6. We start this chapter by showing the security types that represent the policies to be enforced, and the language syntax and semantics. Then, we define the security errors which can be syntactical errors or type errors; develop an algorithm for statically checking syntactical correctness and then present our type system which will check for both syntactical errors and type errors. We prove the soundness of our type system and provide a type reconstruction algorithm and prove its soundness and completeness.

Chapter 6: Future Extension and Discussion In this chapter, we discuss the extensions required to specify and enforce the other policies identified in Chapter 4. Firstly, we extend our security types in Chapter 5 with additional security types that represent policies for the different access types, and extend our type system accordingly to perform additional checks to enforce these policies. We then show how the label structure of the Decentralised Label Model can be adopted, with a slight modification, to incorporate our security types to specify the various policies and how they can be enforced by our type system. We end this chapter by comparing our approach with others existing in the literature. Finally, the thesis is concluded in Chapter 7.

Chapter 2

Background and related work

This chapter discusses related work and provides an overview of file sharing, communication security, perimeter security, insider threats, access control and information flow control.

2.1 Introduction

In this chapter we review topics and related work discussed throughout this thesis. We consider three research areas which are related to our work: (a) research that discusses the activity of file sharing; (b) research that discusses insider threats and (c) research that discusses access control and information flow control, in particular type-based approaches for information flow control.

Our aim in this thesis is to protect the shared files against users who are authorised to access them. Such authorised users are referred to as *insiders* in the literature. We investigate research that focuses on the activity of file sharing to analyse how the sharing activity is performed by different individuals. Such analysis is useful to consider when designing a protection mechanism that will not interfere with people's practices of file sharing. We show that, despite the valuable answers the previous work provided to fundamental questions such as with whom the file is shared, what type of file is shared and how the file is shared and protected, they do not provide an answer to a significant question, which is how files can be propagated and accessed after propagation. Answering this question leads to better understanding of the activity of file sharing, and thus, to designing a protection mechanism that will not prevent users from sharing their files as they desire. Chapter 4 characterises the activity of file sharing based on the answer to this question.

We investigate research that focuses on insider threats in order to analyse the threats imposed by insiders during the activity of file sharing; as it is such threats that we are aiming to prevent in this thesis. The literature on security is divided into three fields: communication security, perimeter security, and insider security. We define each field; present the protection mechanisms developed for it and show that the literature on insider security is lacking a clear definition of what an insider is and what the insider threats are. Consequently, we propose a classification of the insider problem in Chapter 3. Based on the proposed classification, we define the insider and the insider problem, and focus on one category that is related to file sharing.

Protection mechanisms to counter insider threats can generally be divided into detection and prevention approaches. Our interest is in prevention approaches which can be in the form of access control or information flow control. We review the literature on these approaches and focus on a type-based approach for information flow control which is the approach that we adopt to tackle our particular class of insider problem in Chapter 5.

The rest of this chapter is organised as follows: in Section 2.2 we give an overview of file sharing. We provide a definition of the activity of file sharing, show the history of evolving file sharing methods and review previous work that investigate people's practices of file sharing. In Section 2.3 we give an overview of information security. We show the security goals, services, and mechanisms of information security and discuss the security tools developed to secure information in communication, perimeter, and insider security. In Section 2.4 and Section 2.5 we give an overview of access control and information flow control respectively, and focus on a type-based approach for information flow control. Finally the chapter is summarised in Section 2.6.

2.2 File sharing

Most of the existing research on file sharing is focused on specific domains and applications while little research has studied file sharing more broadly [130]. Despite the fact that file sharing is a common activity, few studies of file sharing practices exist in the literature [124]. The majority of research is focused on peer-to-peer file sharing [18, 38] and role-based access control of shared resources [124], while others are focused on personal file sharing, particularly, in the domains of music [123] or photography [3, 71], or professional collaborations in corporations [26].

However, the term *file sharing* has been rarely defined in the literature, and where defined, the definition is tailored to a specific method of sharing; in other words, the activity of file sharing is often defined implicitly by defining the method of sharing under consideration. For examples, in [3, 71] the activity of file sharing is defined implicitly by defining Flicker, a popular photo-sharing website, in [123] by defining iTunes, a digital jukebox software for organising, sharing and listening to music, in [18] by defining Napster, a popular peer-to-peer application for sharing music and in [38] by defining Kazaa, a popular peer-to-peer application for sharing different types of files. However, in these papers and many others, a comprehensive definition of the activity of file sharing that is not related to a particular method of sharing does not exist.

The Oxford English Dictionary defines file sharing as "the practice of making computer files available to other users of a network, in particular the illicit sharing of music and video via the Internet". Although this definition describes the activity of file sharing clearly, it is not completely comprehensive, as it confines the activity of sharing files to those carried out by sharing methods that allow the sharing of files via the Internet and excludes other physical methods of sharing files such as USB.

We are only aware of one study that defined file sharing comprehensively without reference to any specific sharing method. The study is by Whalen et al. [130] who defined file sharing as "the activity of making specified file(s) available to an individual or group, with the option of granting specific right (e.g., ability to view, edit, delete) over those files". The authors identified four key elements in this definition which are the parties who are sharing files (individuals and groups); the files themselves; the means of making files available; and the rights over those files.

Although this definition is more general than the previous one, it has the following drawbacks. Firstly, it confines the type of recipient in the file sharing activity to be either an individual or group, whereas it could be several groups or an unbounded group (i.e. all Internet users). Secondly, this definition makes no distinction between the physical and digital activities of file sharing. However, such a distinction is of great importance as each of these activities constitutes a unique field that has its own methods of sharing and security tools and techniques. Thirdly, this definition makes no distinction between sharing a file by lending a device which contains the file to others and sharing a file by lending the file itself, which is then accessed by others using their devices. The former is considered device sharing rather than file sharing as the device could contain software and hardware to be shared with the file. The latter is considered file sharing as only the file itself is shared.

The general definition of file sharing that we seek, should satisfy the following three properties: Firstly, a clear distinction between physical and digital sharing should be made in the definition. Secondly, the definition should not be restricted to a particular method of sharing files. Thirdly, the definition should not restrict the type of recipient. Fourthly, the definition should exclude those sharing activities performed by sharing the device which contains the files with others. Therefore, we refined the definition by Whalen et al. [130] to satisfy these four properties and define file sharing as follows:

Definition 2.2.1. *File sharing is the activity of making specified digitally stored file(s), (e.g. text, photo, video, audio or software) in a particular device available to others (e.g. an individual, group, groups, or the public) to be accessed by their devices with the option of granting specific rights (e.g. viewing, editing, deleting) over those files.*

This definition excludes sharing files by lending a device which contains the files to be shared with others as well as the sharing of physical files. However, it includes all the methods of sharing digital files, as well as the different types of files and recipients. The person who makes the files available to others will be referred to as *the sender* who is in most cases the owner of the files; while the others, who the files are made available for, will be referred to as *the recipients*. The methods that allow the sender to make the files available to the recipients will be referred to as *the file sharing methods* which can be of various types.

In the next section, we look at the history of file sharing and how the activity of file sharing has been increased over the years by the rapid evolution of file sharing methods.

2.2.1 The history of file sharing

Sharing files is an activity that has been around almost since the infancy of computers. The prevalence of file sharing activity nowadays is attributed to the existence of a variety of methods that simplified this activity. These methods have gone through several stages until they reached maturity at the present time to become fundamental to any Internet user.

Initially, no actual storage media existed; and the only way to transfer information from one computer to another was to type it in manually. Afterwards, the first magnetic storage media which could contain data emerged; however, moving around this magnetic storage was very difficult [24]. The first time file sharing became an easy task to perform was in 1971 when the 8-inch floppy disk was developed by IBM [53, 111, 79, 134]. Although this method of sharing allowed files to be shared easily, the spreading of files went slowly, as the files had to be moved physically from one place to another. In 1978 Ward Christensen created a new method of file sharing which was the first online bulletin board system

(BBS) which allowed users to share files online by utilising their phone lines [134, 135]. A year later in 1979, Tom Truscott and Jim Ellis at the University of North Carolina at Chapel Hill and Duke University created another file sharing method which was Usenet. The main goal of creating Usenet was to facilitate focused discussion threads within topical categories (Usenet newsgroups); however, the transfer of files was a feature of Usenet that users took advantage of [134, 135, 62]. Usenet is considered to be the first network in which users could share files with many other, unknown users [24].

Six years later in 1985, File Transfer Protocol (FTP), which allows files to be efficiently uploaded and downloaded from a central server, was standardised. FTP is still used today as one of the most popular methods of file sharing among individuals and corporations [111, 134, 135]. This was followed by the creation of Internet Relay Chat (IRC) in 1988 by Jarkko Oikarinen which allows users to chat in real-time as well as exchanging files via a Direct Client-to-Client protocol.

A milestone in file sharing occurred in 1990 when the World Wide Web was formally proposed by Tim Berners-Lee and Robert Cailliau [134]. During the nineties the World Wide Web grew to become the largest file sharing network ever created [111]. In 1995, Mosaic which is a graphical internet browser was created and brought more users to the Internet through exciting visuals. Consequently, more information was published, accessed and shared. In 1996, the Multi-Purpose Interment Mail Extensions (MIME) was standardised which allows users to exchange files with each other via email [137]. In 1999, Napster was created by Shawn Fanning and quickly became one of the most popular file sharing methods in the history of computing. Napster is an unstructured centralised peer-to-peer system and is generally cited as the first peer-to-peer file sharing system. However, in 2001 Napster was shut down due to copyright infringement [24, 111, 79, 134, 135].

From 2000 up to the present time, a wide variety of peer-to-peer file sharing systems emerged such as Guntella, eDonkey 2000, Kazaa, BitTorrent as well as web-based file sharing services such as Dropbox, GoogleDocs, youSENDit, Streamfile, Wikisend, 4shared and social networking sites such as Facebook, YouTube, Instagram and Flickr.

In the next section, we look at the previous studies that investigated people's practices of file sharing in order to find answers to several fundamental questions; such as what file sharing methods people utilise, who people share files with, what type of files people share and how people protect their shared files.

2.2.2 Methods of file sharing and people's practices

There are a wide variety of file sharing methods that exist today, each of which has particular features which make them suitable for specific purposes. In other words, based on the features of the methods, a user selects the appropriate one for his sharing situation. These methods range from peer-to-peer sharing applications like Napster, Guntella, and cKaZaA to email, the web, various shared folder systems, application-oriented tools like iTunes and Groove, and web-based sharing tools like BSCW, Wikis and Flicker. Although, many methods exist for sharing files, they all perform the same basic process that requires the users to specify the following information: what should be shared, with whom it should be shared, and how that sharing will take place. However, they differ from one another in the ways of allowing the users to control the what, how, and with whom to share [124].

Olson et al. [82, 81] conducted a pilot study and a more formal survey to explore preferences for general information sharing by investigating what information people are willing to share and with whom. Their findings indicated that people's willingness to share differs from one another, and it depends on who they are sharing the information with; therefore, a one-size-fits-all permission structure for sharing is inappropriate. The authors found that people deal with particular types of information similarly when assessing whether or not to share it with others (examples of categories include, work email and telephone number, pregnancy, health information, email content and credit card number). Also, they found that people deal with particular types of individuals similarly when assessing whether or not to share information with them (examples of categories include, spouse, manager, trusted co-worker, the public and competitors).

The authors believe that their findings can provide guidance on the design of access control and interfaces, that could simplify the policy specification process to the end user. For example, users could be allowed by a preference-specification tool to specify their permissions generally per category of person (e.g., the public, high level people in your organization, co-workers, your family, your manager, your spouse, etc.), while making an exception for one particular person or a particular information type. Furthermore, augmenting such preference-specification tool with some content analysis to detect people's email addresses, social security numbers, or personal facts in the documents, will allow the appropriate permission to be set automatically. For example, people requesting access to a file will be identified by the tool and then based on who they are or which category they belong to they might be denied or granted access.

Voida et al. [124] conducted a survey and follow-up interviews at a medium-sized research organisation to explore users' current practices and needs around file sharing. The authors stated that the understanding the what, with whom and how of sharing will lead us to understand users' current file sharing practices. The results of their study indicated that almost a third of the respondents shared files with groups or classes of individuals, and in many cases these classes mapped directly onto the categories identified by Olson et al. [82, 81]. Also, their survey respondents reported sharing files at work regularly with an average of 7 individuals or groups. With respect to the types of files shared, their respondents reported they shared 34 different types of file or electronic information, which ranged from business documents and paper drafts to music, ideas, schedules, and TV shows. In terms of how the sharing took place, they found that email was the most common method used for sharing files by their respondents (43% of all responses), followed by shared network folder (16%), followed by posting content to a website (11%).

Additionally, the authors point out that all of their survey respondents expect to apply read or full control privileges to their shared files, except for cases where the sharing method offers a set of particular privileges (e.g., iTunes allows sharing recipients to use but not duplicate shared music). Also, their findings indicated that there are three main classes of difficulties and breakdowns that people encounter in sharing, which are: forgetting what file had been shared with who; difficulties in selecting a sharing method with desired features that was also available to all sharing participants; and problems in knowing when new content was made available. Their respondents usually fell back on using the most universal method, which is email, in order to share their files when they were uncertain about the tools available to their intended recipients.

Based on their findings, they identified a number of critical characteristics of file sharing methods including universality, addressing, visibility, notification, and the differentiation between push-and-pull-oriented sharing. Push-oriented sharing is described as actively pushing the file from the sender to the recipient (e.g. email), while pull-oriented sharing is described by simply making the file available for it to be retrieved or pulled at the recipient's convenience (e.g. a shared folder). The authors also developed a prototype of a set of user interface features called a sharing palette which provides a platform for exploration and experimentation with new modalities of sharing. The sharing palette provides a simple and fast way for users to specify the visibility of and permissions for files without the need to maintain access control lists. Also, it provides a various notifications features, which are designed to promote users' awareness of the files they have shared with others.

Whalen et al. [129] conducted an online survey and follow up interviews at a medium-sized industrial research laboratory to address the issue of users' experience of file sharing and access control by gathering information on how and why people share files; the type of information shared; and how, when and why people limit access to those files. The results of the survey showed that email attachments were the most commonly used method for sharing files (98% of all responses), followed by network files sharing (55%), followed by commercial content management system (25%) and removable media (25%). Also their results indicated that 37% of respondents protect their shared files from friends and colleagues, and the methods used for restricting access to their sensitive files are: passwords; permissions/access control lists; physical controls (e.g.,safeguard in office or on person); encryption; obscurity (e. g, giving files innocuous names and hidden directories); and deleting/relocating sensitive files. Based on the results of the study, the authors suggest guidelines to improve methods for appropriate content protection (Table 2.1).

Whalen et al. [130] conducted a web-based survey at a medium-size university to investigate the fundamental issues regarding how files are shared and the difficulties encountered when managing files in collaborative environments. They explored the problem by surveying a group of people regarding the extent of their file sharing, their use of different sharing methods, and the problems they encountered with file sharing in their personal and professional lives. From the results of their survey, they found that file sharing is a common activity, with over 70% of respondents sharing professional and personal files at least once per week. The file sharing methods used by their respondents were email attachments, physical devices (e.g., USB token, CD), networks file share, instant messenger (e.g., MSN, Yahoo), Web server (e.g., webpage, wiki), peer-to-peer (e.g., KaZaa) and file copy protocols (e.g., scp,ftp). The most commonly used file sharing method was email (42.7%) followed by network file share (14.7%) followed by peer-to-peer and file copy protocols (10.3%). This corresponds with the findings of Voida et al. [124] and those of their previous study [129].

Their results also show that there are a number of positive and negative factors that have an impact on people's choice of file sharing methods. The positive factors are: the convenience and the ease of use of the method, the widespread availability of the method in order to reach all recipients and the suitability of the method for the organisation or task at hand. The negative factors are: the limit on file space or file size, lack of access control or security features and the inability to reach all recipients. Furthermore, the results show that the majority of respondents share files between two and four groups, and 80% of respondents have sensitive files. These sensitive files were shared as the results indicated that 44% of respondents shared sensitive professional files and 11% of respondents shared sensitive personal files such as financial or medical information. The authors found that people utilise various methods to control access to their sensitive files, some are technical (passwords, permissions) and others are socially controlled, such as hiding files.

Unlike the study of Voida et al. [124] and Whalen et al. [130, 129] which focused on subjects within a single organisation, all of whom had access to similar, established file sharing methods. Dalal et al. [26] conducted in-depth interviews with respondents across various domains in their homes, home offices, or in cafes where people worked to examine how file sharing and access controls are used, not used or circumvented in order to get work done. The results of their study show that 80% of respondents shared files with overseas collaborators or clients in Europe and the Asia-Pacific region and 100% shared files with colleagues across the US. Their results also showed differences between personal and professional sharing, as they found that in professional sharing, people concentrate on sharing files that are related to project work, such as shared documents including technical specifications, meeting minutes and action items, proposals and reports. On the other hand, they found that in personal sharing, people concentrate on sharing their experiences with others, and the content being shared (primarily multimedia) is relational in nature, such as sharing photographs with family members who live overseas. Email was used by all respondents in their survey, and 80% of them used various social software such as wiki, blogs, social networking sites (including MySpace and Facebook), public websites for sharing images and multimedia files (including Flickr and YouTube), and online forums and games.

Moreover, their respondents made distinctions between two types of sharing, namely sharing with oneself and sharing with others. Sharing files with oneself is very useful as it allows the individual to synchronise activities regardless of location, accessibility, or what devices are at hand. They found that USB drives and email are considered convenient and are the preferred methods for sharing with oneself. Based on their analysis of their results, they derived a set of design criteria for a more effective file sharing system (Table 2.1) [26].

In contrast to previous studies which have focused on asking users themselves to report on how they share and protect files, Smetters and Good [110] conducted an automated survey of access control in a medium-sized corporation to collect behavioural data over time by analysing digital records of actual user behaviour as they believe that users' selfdescriptions of their own behaviour can be incomplete or inaccurate. They used automated data mining to examine how users in a medium-sized corporation utilised two common access control features: the definition of access control groups, and the permissions settings, or ACLs, that users set on folders and documents. They found that access control policies which are applied by users to their content are quite complex. Based on the results of their study, they derived a number of suggestions for the design of both access control systems themselves, and the interfaces used to manage them (Table 2.1) [110].

Mazurek et al. [65] conducted semi-structured interviews with 33 non-technical computer users in 15 households to examine the current practices, needs and attitudes to access control of home users when they share files inside and outside their homes. They found that people utilise a wide range of measures to restrict access to their files, some of them are standard access-control tools while others are ad-hoc tools. These tools are the same as those reported in [129] which are user accounts, passwords, encryption, limiting physical access to devices, and hiding and deleting sensitive files. Also, They found that people have complex policies that change continuously over time, and which are inadequately addressed in current file sharing and access control methods; a finding supported by Olson et al. [82, 81], Whalen et al. [129, 130], and Voida et al. [124]. Based on the results of their study, the authors have generated several guidelines for developers of access control systems aimed at home users (Table 2.1).

Hart et al. [45] surveyed 23 blogging and social networking sites such as Blogger, Facebook, Flickr ,YouTube, and MySpace to determine what access control and privacy features are currently available. They found that a lot of content-sharing sites provide primitive access control mechanisms which make a file entirely private or public while others allow more flexible control by offering a private/friends/public access control model. The authors asserted that these models failed to support people's needs, and thus, proposed a method of access control for content-sharing sites that specify access control polices in terms of the content being mediated. For example, "Blog posts about my home-town are visible to my high school friends". Therefore, based on the posts' contents, the system should automatically specify the policy rules for that post. The authors advocate policy rules that can be applied automatically.

Whalen et al. [131] pointed out that a potential solution for file sharing problems, such as exposing sensitive files accidentally, is to provide the user with clear information about file sharing settings and activities. Therefore, they explored existing research on awareness in collaborative environments, and used it to develop a framework for file sharing awareness. The authors used this awareness framework to develop a prototype for a file manager that facilitates file sharing by making sharing activity and settings more visible to the user. The file manager displays file sharing activities such as sends and accesses as icons with files details and permissions. Also, it includes a sharing console that allows users to view more detailed sharing information about a specific file, to search for files matching certain criteria, and to set and view alerts on shared files.

Table 2.1 summarises the recommendations of the previous studies for the design of access control systems. These recommendations generally fall into three categories, which are visibility, usability, and suitability recommendations as shown in Table 2.2.

Authors	Guidelines and Recommendations for Access Control System Design
Whalen et al. [129]	1. Fit access control management into the user's task. 2. Make access control decisions visible. 3. Make the controls themselves simple to manage. 4. Support, rather than replace, social controls. 5. Design for sharing across organisational and file system boundaries. 6. Allow users to choose from a palette of sharing and security tools.
Dalal et al. [26]	1. No impedance matching: (a) The system should work for all types and sizes of data. (b) Responders in particular should be required to have no more than minimal, readily available tools (e.g. email and a web browser). 2. Support ad-hoc sharing: (a) Use universal identifiers, such as email addresses; people should be able to share with anyone, inside or outside of their organisation, with equal facility. (b) Minimise setup effort as users will not know upfront whether they will share with a particular group or use a specific mechanism enough times to make the effort worthwhile. (c) Require no a priori preparation by responder. 3. No oversharing: (a) Content shared only with intended recipients. (b) Transient access management. 4. Simple and self-contained: (a) Interactions should be lightweight and familiar. (b) One-step sharing (i.e. additional coordination, such as follow-up emails should not be necessary).
Smetters and Good [110]	- Simplify access control models: 1. Only allow positive grants of access. 2. Simplify the inheritance model for access control changes 3. Limit the types of permissions that can be granted. 4. Group Definitions. -Improve Tools For Managing Access: 1. Tools for group management that reduce redundancy and error in group definitions, and track the intended relationship between groups 2. Tools for ACL management that maximise the use of groups, and help generate concise ACL statements granting only necessary rights. 3. Tools for administrators to manage access policy, directly focused on "cleaning up" outdated users, groups and permissions. 4. Activity-based folksonomies of groups and users to help users choose the right principals with whom to share among potentially similar groups, and make old groups "fade away" naturally. 5. Visualisations to enable users to see who has access to the content they are sharing, and what content is impacted by a change in policy.
Mazurek et al. [65]	1. Allow fine-grained control. 2. Plan for lending devices. 3. Include reactive policy creation. 4. Include logs. 5. Reduce or eliminate up-front complexity. 6. Acknowledge social conventions. 7. Support iterative policy specification. 8. Account for users' mental models.

Table 2.1: Recommendations for the design of access control systems

Visibility recommendations such as making access control decisions visible to users, is a very useful concept design for access control systems. It promotes users' awareness of the sharing activities, and thus reducing the possibility of accidental exposure of sensitive files as indicated by Whalen et al. [131]. One of the main classes of difficulties people encounter in sharing, pointed out by Voida et al. [124], is forgetting what file had been shared with who. Therefore, by making the activity of file sharing visible, mistakes made during the activity of file sharing can be identified easily and rectified immediately (e.g. revoke permissions granted to unauthorised users accidentally). Usability recommendations such as minimise setup effort and reduce or eliminate up-front complexity is another concept design for access control systems, which contribute to the availability of the method. This is because one of the positive factors which have an impact on people's choice of file sharing methods is the convenience and ease of use [130]. Therefore, complicated methods will be less preferred to be used by users, and thus its availability will be reduced among participants. Suitability recommendations such as design for sharing across organisational and file system boundaries, and content shared only with intended recipients is the third concept design for access control systems, which also contribute to the availability of the method. A method is suitable if the method provides people with all the desired features they need to accomplish their tasks. These features can be divided into features that facilitate information sharing such as allowing files to be shared across organisational boundaries, and features that secure information sharing such as disallowing files to be shared with unintended recipients. Both kinds of features must be considered by the sharing method to be suitable. For example, lack of the former features lead to a file sharing method that is secure but does not allow files to be shared as people prefer, while lack of the latter features lead to a file sharing method that allows files to be shared as people prefer but is not secure. Therefore, considering one kind of features and ignoring the other will result in a file sharing method that is not suitable.

Although the three categories of recommendations mentioned above are equally important and must be considered when designing access control systems, in this thesis we are only concerned with the suitability of the method for the task of sharing. This is because our aim is to protect the shared files, and thus we must ensure such mechanism is suitable for different tasks of sharing and does not interfere with people's practices of the activity of file sharing. However, the suitability recommendations shown in Table 2.2 are quite general. For example, fitting access control management into the user's task and content should be shared only with intended recipients require the task and recipients to be known in advance. To design a suitable protection mechanism, we need to investigate people's practices of the activity of file sharing. Such investigation is a very useful step towards characterising the activity of file sharing into different categories. These categories can be thought of as policies that must be enforced. In this way, the enforcement mechanism will not only protect the shared files, but also allow users to share their files as desired.

Recommendations	
Visibility	1. Make access control decisions visible. 2. Visualisations to enable users to see who has access to the content they are sharing, and what content is impacted by a change in policy. 3. Include logs.
Usability	1. Make the controls themselves simple to manage. 2. Use universal identifiers, such as email addresses; people should be able to share with anyone, inside or outside of their organisation, with equal facility. 3. Responders in particular should be required to have no more than minimal, readily available tools (e.g. email and a web browser). 4. Minimise setup effort as users will not know upfront whether they will share with a particular group or use a specific mechanism enough times to make the effort worthwhile. 5. Require no a priori preparation by responder. 6. Interactions should be lightweight and familiar. 7. One-step sharing (i.e. additional coordination, such as follow-up emails should not be necessary). 8. Only allow positive grants of access. 9. Simplify the inheritance model for access control changes 10. Limit the types of permissions that can be granted. 11. Group Definitions. 12. Tools for group management that reduce redundancy and error in group definitions, and track the intended relationship between groups 13. Tools for ACL management that maximise the use of groups, and help generate concise ACL statements granting only necessary rights. 14. Tools for administrators to manage access policy, directly focused on "cleaning up" outdated users, groups and permissions. 15. Activity-based folksonomies of groups and users to help users choose the right principals with whom to share among potentially similar groups, and make old groups "fade away" naturally. 16. Reduce or eliminate up-front complexity.
Suitability	1. Fit access control management into the user's task. 2. Design for sharing across organisational and file system boundaries. 3. Allow users to choose from a palette of sharing and security tools. 4. The system should work for all types and sizes of data. 5. Content shared only with intended recipients. 6. Support, rather than replace, social controls. 7. Transient access management. 8. Allow fine-grained control. 9. Plan for lending devices. 10. Account for users' mental models. 11. Support iterative policy specification. 12. Include reactive policy creation. 13. Acknowledge social conventions.

Table 2.2: Categories of recommendations

Previous studies reviewed in this section provided valuable answers to fundamental questions that could lead to unsderstand the activity of file sharing, such as: with whom is the file shared, what type of file is shared, and how the file is shared and protected (Table 2.3). They found comprehensive results in terms of knowing with whom people share their files, what type of files they share, and how they protect their shared files. However, we believe that the question of how the file is shared has not been answered or has not been answered properly. They merely answered the question of how people share their files by enumerating the methods of sharing files that people utilised. Such an answer applies only to the question of what methods people utilise to share their files rather than how the files are shared.

Therefore, in Chapter 4, we investigate the question of how people share their files by characterising the activity of file sharing based on two factors: how files are propagated, and how files are accessed after their propagation. Our characterisation of the activity of file sharing results in different categories of sharing activity that can describe all possible ways of how users share files with each other.

	With whom the file is shared	What type of file is shared	How the file is shared	How the file is protected
Olson et al. [82]	-The public, co-workers, managers and trusted co-workers, family and spouse.	-Email content, credit card number, transgression, work related documents, work email and desk phone number.	-	-
Voida et al. [124]	-Similar to Olson et. Al.-With an average of 7 individuals or group	-34 different types of files e.g. business documents, paper drafts, music, ideas, schedules, and TV show	-Email (43%), shared network folders (16%) and posting content to a web site (11%)	-
Whalen et al. [130]	-Over 69% shared with two to four groups such as friends, family, research group, general public and colleagues. -25% shared with five to twenty groups.	-Only focused on sensitive files, such as email, personal financial or medical information, professional data or documents of an organisation, professional data or documents governed by law.	-Email (42%), shared network folders (14.7%), peer-to-peer program (10.3%) and file copy protocol (10.3%)	Various methods to control access to their sensitive files, some are technical (passwords, permissions) and others are socially-controlled such as hiding files.
Whalen et al. [129]	-	-	-Email (98%), shared network folder (55%), commercial content management systems (25%) and portable devices (25%)	Passwords; permissions/ access control lists; physical controls (e.g., safeguard in office or on person); encryption; obscurity (e.g., given files innocuous names, hidden directories); and deleting/relocating sensitive files.
Dalal et al. [26]	-With employees in professional sharing -With friends and family in personal sharing.	-In professional sharing: revolve around project work such as technical specifications, meeting minutes, and action items, proposals, reports.-In personal sharing: revolve around multimedia relational in nature such photograph and video.	Email (100%), - 80% used a wide variety of social software, such as wikis, blogs, social networking sites (including MySpace and Facebook) hosted services (such as Yahoo! Briefcase) public websites for sharing image and multimedia files (including Flickr and YouTube) and online forums and games.	-
Mazurek et al. [65]	-Family, friends, co-workers and strangers.	-Music, photo, video, private documents, school work, work files, and other personal documents.	-	-User accounts, password, encryption, limiting physical access to devices, and hide and delete sensitive files.

Table 2.3: Summary of previous studies on file sharing

2.3 Security

Due to the widespread sharing of digital information and the rise in threats associated with it, the security of digital information has become one of the biggest concerns for governments, corporations and ordinary individuals; each of which is searching for tools to protect their sensitive information. As a result, the field of information security has become one of the hottest topics in the recent past. However, information security is much more than just protecting digital information sharing, it means "protecting information and information systems from unauthorised access, use, disclosure, disruption, modification, or destruction in order to provide confidentiality, integrity and availability" [101]. Therefore, information security is not only concerned with protecting the information itself, but also with protecting each component of the information system. The components of an information system are the entire set of software, hardware, data, people, procedures, and networks [132]. Different components may require different types of protection which can be divided into technical protection, physical protection, and awareness-based protection. For instance, software, data, and networks are protected by technical protection such as encryption and firewall. Hardware is protected by physical protection such as locks and keys that restrict access to the area where the hardware components are located. People and procedures are protected by awareness-based protection such as educating and training individuals to prevent them from accidental abuse of information; alternatively, they could be protected by technical protection to prevent individuals from intentionally misusing information.

Hence, in order to protect digital information, each component of the information system must also be protected as well by a combination of physical, technical and awareness-based protections. However, in this thesis we are only concerned with technical protections. In the next section, we look at the goals of information security and the security services.

2.3.1 The goals of information security

As mentioned above, the goals and the concept that underlie information security are to achieve confidentiality, integrity and availability of information. *Confidentiality* means preventing unauthorised disclosure of information. *Integrity* means preventing unauthorised modification of information. *Availability* means preventing unauthorised withholding of information so that information should be accessible and usable upon appropriate demand by an authorised user. It can be seen that each goal is met by allowing only authorised people to perform the action while disallowing the actions of unauthorised people. For instance, confidentiality is met if only authorised people can view or access the information; integrity is met if only authorised people can modify the information; and availability is met if only authorised people can withhold the information. Hence, in order to achieve these goals, there must be a method to differentiate authorised from unauthorised people. To solve this issue, three steps must be performed as follows: Firstly, each person must have a unique identity to identify him/herself to the system; this step is known as *identification*. Secondly, each person must prove that he/she is really who they claim to be to the system; this step is known as *authentication*. Finally, the system must define what the authenticated person can or cannot do with the information; this step is known as *authorisation*. These three steps constitute what is called *access control* which in turn provides the confidentiality, integrity and availability of the information. Access control is discussed in more detail in Section 2.4.

The three main goals of information security which are confidentiality, integrity, and availability along with authentication, authorisation (or access control), and non-repudiation (which means preventing a person from denying later that he/she participated in a transaction) are security services that are defined by ITU-T Recommendation X.800 [19]. X.800 defines a security service as "a service provided by a protocol layer of communicating open systems, which ensures adequate security of the systems or of data transfers". A clearer definition is found in RFC 2828, which defines a security service as "processing or communication service that is provided by a system to give a specific kind of protection to system resources; security services implement security policies, and are implemented by security mechanisms" [106]. Security mechanisms are techniques designed to detect, prevent, or recover from a security attack. The aforementioned security services are implemented through various security mechanisms as there is no a single security mechanism that can provide all the security services.

There are a wide variety of security mechanisms, that each of which provides different security services, and many of which are based on cryptographic techniques. For instance, encryption is used to achieve information confidentiality, whereas hash algorithms are used to achieve information integrity. A digital signature, based on public key cryptography, is used to achieve non-repudiation by identifying the source of the information. Authentication can be achieved through public key cryptography, where the public key of the key pair can be signed by a trusted third party, often called the Certificate Authority (CA), and becomes an electronic authenticated identity for a specific person or organisation. Although authenticating identities over networks, which can be seen as machine-to-machine authentication, is necessary; human-to-machine authentication is of great importance as well. This type of authentication can be achieved by various mechanisms which range from something the user knows (such as password), and something the user has (such as token devices and smart cards) to something the user is (such as biometrics).

Despite the fact that each security mechanism provides one or a few security services, real world scenarios often require combination of multiple security services working together to meet specific security goals. Consequently, available security tools seek to combine several security mechanisms to provide multiple security services that satisfy particular requirements. One example of this is access control. The term *access control* is often used as a synonym for authorisation. However, in this thesis, we define access control broadly as a tool that requires multiple security services which are implemented by several security mechanisms to satisfy a particular goal which is controlling access to and interaction with system resources. The security services required by access control are confidentiality, integrity, availability, authentication, authorisation, and occasionally non-repudiation. As a result, many mechanisms are used in access control to achieve these security services. Other examples of such security tools are cryptographic protocols, intrusion detection systems, and firewalls. It must be taken into account that these tools can be divided into either intrusion prevention or intrusion detection tools. The former are tools that prevent an attack while the latter are tools that detect an attack; and they complement rather than replace each other. For instance, once an intrusion prevention tool fails to prevent an attack, an intrusion detection tool comes in to play an important role in detecting the attack and taking another action.

The wide range of security tools that exist today is the result of extensive efforts in the security literature to counter various attacks. Generally speaking, the literature on security can be divided into three fields which are (i) communication security, (ii) perimeter security, and (iii) insider security. Each of these fields has developed security tools to counter particular types of attack. In the next sections, we review each field and focus on insider security as it is the most relevant to our work. Figure 2.1 illustrates the three fields of security and their domains.

2.3.2 Communication security

The literature on communication security is concerned with preventing different types of attacks on data transmitted over a network. In other words, the field of communication security deals with attacks that target the communication link between two entities that are

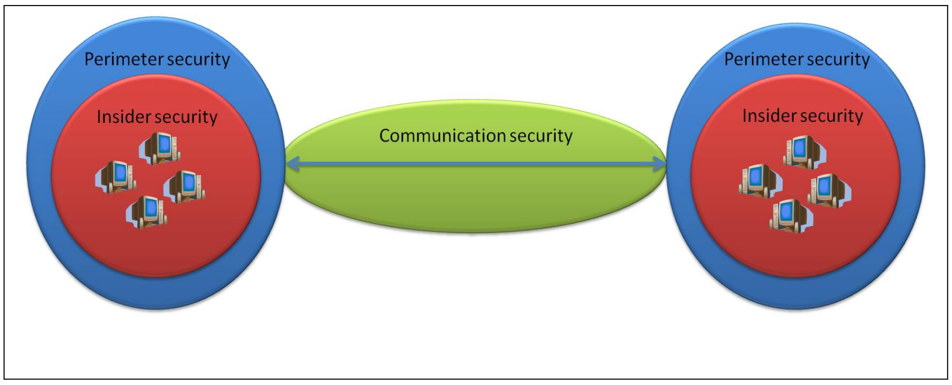

Figure 2.1: Security fields

sending and receiving data. The purpose of these attacks is to violate the confidentiality, integrity, and availability of data transmitted over networks.

Most of the work in communication security is dedicated to developing tools to protect the transmission of data over a network [108, 41, 57, 36, 104]. Such tools are known as cryptographic protocols which utilise different cryptographic mechanisms to provide security services which in turn counter the different security attacks. Cryptographic protocols (also known as security protocols) are implemented at different layers of the network architecture. For instance, PGP [140],S/MIME [93] and Kerberos [10] are cryptographic protocols at the application layer; SSL/TLS [36] at the transport layer; IPsec [57] at the network layer; PPTP [41] at the data link layer, to name but a few. The reason behind implementing cryptographic protocols at different layers of the network architecture is that cryptographic protocols at one layer offer different degrees of protection from cryptographic protocols at another layer [127]. As a consequence, there exist many cryptographic protocols implemented at different layers in order to protect network resources from different types of attacks [78, 39].

The cryptographic protocols implemented at the different layers of the network, play an important role in protecting the shared files. Due to the speed and ease of use, most of the file sharing activity is performed through networks; hence, for a file to be shared, it must be transmitted over the network to the recipients. Therefore, while the shared file is transmitted, it will be susceptible to attacks if none of the aforementioned cryptographic protocols is implemented. Furthermore, for files sharing activities that are not performed through networks, for example through removable devices, the shared files will be susceptible to loss or theft. Therefore, encryption mechanisms can be of great importance in such situations, so that only authorised users can decrypt the file rather than any one who possesses the removable device. To summarise, these protocols are needed to provide protection to the shared files when they are moved from one location to another.

2.3.3 Perimeter security

The literature on perimeter security is concerned with protecting the data while it is being stored. Unlike the field of communication security which is focused on protecting data transmitted over an untrusted network, the field of perimeter security is focused on protecting data stored in a trusted internal network. In the literature, perimeter is defined as "the fortified boundary of the network" [80], and it is understood as a way of protecting internal networks, which are considered safe, from attacks coming from external networks such as the Internet which is considered unsafe.

Therefore, most of the work in perimeter security is dedicated to developing tools to protect the boundary of the internal network where most of the valuable resources reside, so that attackers cannot get into the internal network and violate the confidentiality, integrity and availability of the stored data. Such tools are Firewall, Intrusion Detection Systems (IDS), and Intrusion Prevention Systems (IPS).

Firewall

A firewall is the most widely used security tool to protect an internal network from external attacks. It is placed between the internal network and the external network as a barrier to determine what traffic can get into or out of the internal network. Firewalls can be seen as an access control for networks that can be used to protect Local Area Networks (LANs), Personal Area Networks (PANs), Wireless Local Area Networks (WLANs), Wireless Sensor Networks (WSNs) or even a single host. According to [116], even though there is no standard firewall terminology, there are three main types of firewalls, each of which examines data up to a specific layer in the OSI reference model. These types are as follows (i) *A packet filter* is a firewall that operates at the network layer. (ii) *A stateful packet filter* is a firewall that operates at the transport layer. (iii) *An application proxy* is, as the name suggests, a firewall that operates at the application layer where it functions as a proxy. Each type has its own advantages and disadvantages. More details about the different types can be found in [113, 127, 116]

IDS

IDS are the second line of defence that protects the internal network from attackers who have already managed to pass through the firewall. As the firewall works at the boundary of the internal network, it cannot prevent malicious activities inside the internal network. Hence, an IDS plays an important role in detecting such attacks that the firewall is not able to prevent. It is usually used as a complementary tool to the firewall such that if the firewall fails to identify or prevent an attack, the IDS will detect the attack and report it to the network administrators. There are two types of IDS which are Host-based IDS and Network-based IDS. The former operates on a single host and monitors traffic at that host by utilising the resources of its host to detect attacks. The latter operates as a stand-alone devices on a network and monitors traffic on the network to detect attacks [116, 80, 56].

Additionally, IDS utilises two methods for detecting attacks which are Signature-based IDS and Anomaly-based IDS. The former detects attacks based on known signatures or patterns which is similar to signature-based virus detection. The latter defines the normal behaviour of a system and reports attack whenever the system behaves abnormally [116, 80, 56].

IPS

IPS is similar to IDS except that IPS is not only able to detect attacks and report it to administrators, but also able to block those attacks when they have been detected without direct involvement of the administrators. Therefore, such tools combine the functionality of a firewall and an IDS to offer detective and preventive solution that block actions which have been detected as an attack [80].

Eventually the shared files will be stored in a location (e.g. a recipient device or central server) to facilitate access to them by the recipients. Such a location will be a target for attacks from the outside. Therefore, the security tools reviewed in this section, can be used to protect the shared files in such situations.

2.3.4 Insider security

The literature on insider security is the most relevant to our work. It is concerned with preventing attacks performed inside the perimeter of the trusted internal network. Although perimeter security prevents network attacks on stored data, other attacks can be performed without using a network connection by gaining access to a local device to view, modify, or destroy the stored data. Also, viruses and Trojan horses can be introduced to a local machine by inserting an affected optical disc into it without the need to propagate them through the networks. Therefore, protecting stored data is an area where network security and computer security overlap [113].

In contrast to communication and perimeter security which deal with attacks performed by external attackers, insider security deals with attacks performed internally by those who are authorised to access the data. The features of being inside the perimeter of the internal network and an authorised person differentiate insider attacks from external attacks and make such attacks difficult to tackle.

According to the 2011 CyberSecurity Watch Survey, conducted by the U.S. Secret Service, the CERT Insider Threat Center, CSO Magazine, and Deloitte [55], 58% of the attacks are caused by outsiders (those who are unauthorised to access network systems or data) while 21% of the attacks are caused by insiders (those who are authorised to access network systems or data), and 21% are from unknown sources. Even though the percentage of insider attacks is less than the external attacks, the consequences of insider attacks can be more severe. The survey indicated that 33% of respondents consider insider attacks to be more costly and damaging. Consequently, insider attacks are a serious danger and should be paid similar attention to that paid to external attacks.

Hunker [50] indicated that there exists a large body of work in the literature to address the insider threats problem; however, a little progress has been made to reduce the insider threat problem. The author attributed the slow progress in the field to the absence of clear answers to fundamental questions. One of these questions is "What is an insider threat?". The author noted that "if we cannot rigorously define the problem we are seeking to solve, then how can we approach it? or even know when the problem has been solved" [50].

The terms *insider* and *insider threat* have been defined in many different contexts by different authors. Some authors have focused on the trust relationship when defining the term insider. For instance, the RAND report [6] defined the insider as "an already trusted person with access to sensitive information and information systems". Bishop [14] defined the insider as "a trusted entity that is given the power to violate one or more rules in a given security policy". Other authors have focused on the abuse of given access privileges. For instance, Chinchani et al. [22] defined the insiders as "legitimate users who abuse their privileges". The CERT report [68] defined the insider as "individuals who were, or previously had been, authorised to use the information systems they eventually employed to perpetrate harm".

Others defined the insider very broadly. For instance, Predd et al. [88] defined the insider as "someone with legitimate access to an organisation's computers and networks". The RAND report [6] defined the insider again as "anyone with access, privilege, or knowledge of information system and services". The former definition might include masqueraders who stole the credentials of a legitimate user to get access to the computer or the network. The latter definition eliminates the need for trust and includes those who have knowledge of the system or the service even if they do not have access privileges.

In 2008, a cross-disciplinary workshop on "Countering Insider Threats" [89] concluded that

> "an insider is a person that has been legitimately empowered with the right to access, represent, or decide about one or more assets of the organisation's structure"

With regard to insider threat, Predd et al. [88] defined insider threat as "an insider's action that puts an organisation or its resources at risk". The RAND report [6] defined it as "malevolent (or possibly inadvertent) actions by an already trusted person with access to sensitive information and information systems". Hunker and Probst [51] defined it as follows "an insider threat is [posed by] an individual with privileges who misuses them or whose access results in misuse". The CERT Insider Threat Center's current definition of insider threats is as follows:

> "A malicious insider threat to an organization is a current or former employee, contractor, or other business partner who has or had authorized access to an organization's network, system, or data and intentionally exceeded or misused that access in a manner that negatively affected the confidentiality, integrity, or availability of the organization's information or information systems". [20]

The CERT definition of insider threats is focused on intentional misuse and excludes accidental misuse. It can be clearly seen that there exists a wide variety of definitions of insider and insider threat. Bishop et al. [15] and Bishop and Gates [16] point out that each author who discussed the insider problem, has made his/her own definition of the insider or the insider threat. This matter has complicated the research in insider threats as one solution to the insider problem might not be applicable to another insider problem. The authors also point out that difficulty in defining the term insider stems from the fact that the perimeter of the organisation network can be defined as well, such that anyone inside the perimeter is therefore an insider. However, with the increased usage of mobile computing, outsourcing and contracting, the concept of a distinct border around an organisation has become blurred.

Due to the differences and contradictory definitions of insider and insider threats that complicate the problem to be solved, many authors are urging the community to establish a framework or taxonomy for distinguishing among different types of insider threats [89, 51, 90]. They mentioned that each determining factor for an insider can be used for a taxonomy, for example based on distinctions between:

- Malicious and accidental threats;
- Doing something intentionally (for malicious or good reasons which nonetheless may result in damage) versus events that occur accidentally.
- Obvious and stealthy acts.
- Acts by masqueraders (e.g, an individual with a stolen password), traitors (malicious legitimate users) and naive or accidental use that results in harm.
- A combination of factors such as access types; aim or intentionality or reason for misuse; level of technical and the system consequences of insiders threats. [89, 51, 90]

Bellovin [11] identified three different types of insider attack which are misuse of access, defence bypass, and access control failure.

- Misuse of access: the insider missuses the system's resources through the privileges he/she was given. This form of attack is the hardest to detect or prevent by purely technical means as the insider already has legitimate access. The best solution is to monitor unusual patterns or quantities of requests, detailed logging can be used when a person falls under suspicion for other reasons.
- Defence bypass: insiders are generally inside the perimeter which means that they are already past some layers of defence. This makes the insider able to commit mischief to the system's resources easily compared to external attackers who need to pass several layers of defence. Also, this form of attack is hard to conceive of in purely technical means. Reliance on technical or non-technical detection of anomalous behaviour or actual attacks is required.
- Access control failure: the insider should not have access to specified system resources. Unlike misuse of access and defence bypass attacks, access control failure attack is a technical problem. While prevention is straightforward, detection of access-control failures is difficult for the same reasons as with access-control misuse [11].

It can be seen that not all of the attacks can be countered by purely technical means; thus, other non-technical means are important to solve the insider threat problem. Hunker and Probst [51] identified three different approaches to solve the insider threat problem and which current works in the field scattered among them. These approaches are the technical approach, the socio-technical approach, and the sociological approach. The authors note that technical approaches are focused on policy languages, access control and monitoring, while socio-technical approaches are focused on policy, monitoring and profiling, prediction, forensics and response work. Sociological approaches are focused on motivation, organisational culture, human factors and privacy and legal aspects.

Silowash et al. [107] analysed cases of insider threat from the CERT insider threat database, which contains more than 700 cases of insider threat, and observed that malicious insider activities can be classified into four classes as follows.

- IT sabotage: an insider's use of IT to direct specific harm at an organisation or an individual. Examples of this are destroying critical data, or planting a logical bomb to delete data at critical times, etc.

- Theft of Intellectual Property (IP): an insider's use of IT to steal IP from the organisation. This category includes industrial espionage involving outsiders. Examples of usually stolen IP assets are proprietary software, business plans, product details, and customer information.

- Fraud: an insider's use of IT for the unauthorised modification, addition, or deletion of an organisation's data (not programs or systems) for personal gain, or theft of information that leads to an identity crime (e.g., identity theft or credit card fraud).

- Miscellaneous: cases in which the insider's activity was not for IP theft, fraud, or IT sabotage.

Technical approaches for encountering insider threats can be divided into detection or prevention tools. The former is based on monitoring while the latter is based on access control. Sinclair and Smith [109] note that most of the work on insider threat space is focused on detection tools. The reason behind this, is that most of the authors believe that insider attacks cannot be prevented as those insiders are operating within their privileges, but rather it is feasible to observe the patterns of information use to detect attacks and respond to them. For instance, with regard to the two types of insider attacks which are misused of access and defence bypass; such attacks are difficult to prevent due to the fact that the insider is not using more than just the privileges he/she has legitimately acquired to perform the attack. However, it is easier to detect anomalous insider behaviour or to monitor an already suspected insider.

Nevertheless, detection tools suffer from several drawbacks similar to those of intrusion detection systems, because insider detection tools utilise the same method of detection (e.g, signature-based and anomaly-based) but with extensions to counter insider threats.

For instance, signature-based detection tools can only detect attacks if their signatures are already known, otherwise the attacks go undetected. Anomaly-based detection tools may lack accurate descriptions of normal behaviours which will be used to define anomalous behaviours. Also, if an attack occurred during the phase of monitoring the normal behaviour, the attack will not be detected later as it will be assumed to be a normal behaviour. Hunker and Probst [51] indicated that the lack of data about insider attacks, made it hard to find out whether monitoring is effective in this space or not. They noted that monitoring is beneficial if an insider attack is already suspected.

Access control as a prevention tool is rarely discussed in the literature to address the insider threat problem. It is usually mentioned to illustrate its drawbacks and to promote detection tools in the insider threat space. However, Sinclair and Smith [109] stated that "better prevention can simplify the problem space that detection must address". The authors surveyed existing research and development in access control, focusing on the applicability of this work in preventing insider attacks in large organisation environments. Although they found that the theory behind access control and the systems that implement it seem to be well-developed, the insider threat problem is still there. Consequently, they raised several challenging questions in terms of access control and the insider threat problem. Such questions are: "Have the basic principles of access control overlooked something?"; "Would a practically correct access control system even reduce the incidence of insider attack?" "Is such an access control system possible?"; "Can all insider threat be prevented with well-designed access control mechanisms?".

From our perspective, the problem is not with the prevention tool, but rather with the ambiguous definitions of the insider problem and threats. For prevention tools to be used effectively to tackle the insider problem, the problem and the threats must be defined precisely. However, a single definition for the insider problem and its threats as attempted by previous work is not suitable. We suggest that the insider problem should be divided into smaller problems that can be defined, studied and solved independently. Consequently, in Chapter 3 we propose a classification of the insider threat problem, and focus on one category that is related to file sharing. We define the threats that are imposed by insiders in this category; as in this thesis we are concerned to prevent threats such as these. Since the threats are precisely defined, we follow a prevention approach to preventing such threats rather than a detection approach.

Generally, a prevention approach falls into two forms which are access control and information flow control. Our interest in this thesis is with information flow control, particularly type-based approach to information flow control. This is because software is the major cause of many breaches in security, and a promising approach to create a secure software is to write it in a typesafe programming language. Therefore, we take a type-based approach to enforce information flow policies which is a language-based technique to provide security in programs. However, information flow control is a complementary approach to access control, since the latter restrict the release of information whereas the former restrict its propagation. Therefore, in the next sections we give a brief overview of access control by showing its components and the advantages and disadvantages of well-known access control models. Then, we focus on type-based information flow control which is the approach that we adopt to tackle our particular class of insider problem in this thesis.

2.4 Access control

Access control regulates access to resources, and has become one of the central themes of security. The major function of access control is to manage the access rights of users when fully sharing the system's resources, and to ensure that illegal uses and access to the network resources cannot occur. Access control limits the access of the subject to the object and controls the subject's access according to its identity authentication. Qing-hai and Ying [92] indicate that access control is an important measure in providing protection for the system's resources; and it is considered the most important security mechanism in a computer system; and one of the most important measures to achieving confidentiality and integrity of data. In the this section, the components of access control are described first, and then different models of access control are discussed in detail.

2.4.1 Access control components

Access control is comprised of three important components, which are identification, authentication and authorisation. Each of which complements the others, and which must be implemented in order. For instance, a subject must first be identified then authenticated and finally authorised to access an object [43]. It is worth taking into consideration that authentication and authorisation techniques play an important role in defining the level of security in an application. Therefore, on the basis of the security level required for each application type, authentication and authorisation techniques must be selected carefully in order to reach the desired level of security. In the following subsections, each of the access control components is discussed in more detail.

2.4.2 Identification

Identification is the first step in access control. Stewart et al. [117] defined identification as “The process by which a subject professes an identity and accountability is initiated”. Hence, the identification process is established when a user provides a user-name, a log-on ID, a personal identification number (PIN), or smart card. Once a subject has identified himself, his identity will be accountable for further actions undertaken by him. Identification is about providing a public piece of information (user-name, account number), and it might be known by a subject’s friends or family [43]. Thus, identification does not play an important role in making the application more or less secure. However, it is only the first step and the starter point that facilitates, introduces, and is relied on by the two most important steps which are authentication and authorisation.

2.4.3 Authentication

Authentication is the second step in access control, and it relies on the identification step. While identification is about providing a public piece of information, authentication is about providing a private piece of information that is known solely by a certain subject [43]. Stewart et al. [117] defined authentication as “the process of verifying or testing that a claimed identity is valid”. They state that in the authentication step, additional information is needed from a subject and it must correspond exactly to the identity professed. A well-known example of authentication is a password.

Harris [43] argues that authentication techniques can be classified intro three types based on their characteristics as follows: Type 1: subject must prove something he knows (e.g. password). Type 2: subject must prove something he has (e.g. smart card). Type 3: subject must prove something he is (e.g. fingerprint). He indicates that authentication techniques having just one of these characteristics are referred to as one-factor authentication. Authentication techniques having two of these characteristics are referred to as two-factor authentication, whereas authentication techniques having all three of the characteristics are referred to as three-factor authentication. Therefore, in order to have a strong authentication process, the authentication should utilise at least two-factors or more. An example of utilising two-factors is when a subject uses a debit card at a shop; he must swipe the card (something he has) and enter a PIN (something he knows) to complete the transaction.

It is worth mentioning that more than one technique can be associated with a one-factor approach of the same type. For instance, in type 3, where a subject must prove something he is, there are several techniques used to achieve the authentication process such as a fingerprint, finger scan, palm scan, retina scan, iris scan and so on. Also, it should be borne in mind that these techniques provide various levels of security, which means that some of them are more reliable, secure and accurate than others. Therefore, the level of security in an application is not only determined by the number of authentication factors used, but also by the techniques used in each type of the authentication process.

2.4.4 Authorisation

Authorisation is the third and final step of access control, it is performed after a subject has been identified and authenticated. Harris [43] defined authorisation as "A process of assigning authenticated subjects access and the right to carry out specific operations, depending upon their preconfigured access rights and permissions outlined in an access criteria". It must be noted that not every identified and authenticated subject can use all resources; and after a subject is identified and authenticated, the subject must be checked to find out what accesses and operations he can perform. In other words, by performing the authorisation step, we can determine what an identified and authenticated subject can actually access and what operations can be carried out.

According to Harris [43] and Stewart et al. [117] authorisation is provided by a system through access control models which manage the type and extent of the subjects' access to objects. An access control model is "A framework that dictates access control using various access control technologies" [91]. Harris [43] indicates that the main objective of access control models is to enforce the rules and objectives of certain security policies and to dictate how objects must be accessed by subjects. There are different types of access control models, and each model has its own advantages and disadvantages. Each of the existing access control models serves different organisational needs, according to their culture, the nature of business, security policy, and so on [91, 117]. Access control models can be broadly categorised into three main categories; namely, *traditional access control models, the trust management model* and *the Digital Rights Management (DRM) model.* In the following sections, we briefly review these three categories.

2.4.5 Traditional access control models

Traditionally, access control came into existence to address the needs of two major fields: the military and the commercial. The former focuses on confidentiality of data, whereas the latter focuses on flexible models for data integrity [25]. These two needs have led to the emergence of two distinct access control models which are *mandatory* and *discretionary* access control models. However, the limitations of these two models of access control has led to further research in this area that has resulted in the emergence of *role-based* and *task-based* access control models [98].

The mandatory access control model (MAC), discretionary access control model (DAC), and role-based access control model (RBAC) are regarded as the most widely accepted access control models [92, 8]. Therefore, they are covered in this section.

Mandatory access control In mandatory access control models, each object is attached to a security label and each user is assigned to a security clearance. Access restriction in this model is based on the security clearance of users and security labels of objects. In order for a subject to access an object, the subject's clearance level must be equal to or greater than the object's label level. For instance, if an object in an organisation is considered very confidential and it has been assigned a "Very confidential" security level, a subject who has been assigned the security level "Confidential", cannot access the object as his/her security level is lower than that of the object. As the name of this model suggests, it does not allow a subject or a program of the subject to modify the security levels, instead they are enforced by the system and only the administrators of the system can modify them. This has led this model to be stricter and more secure than DAC but neither as flexible nor as scalable; and also made this model suitable for applications that require a high level of security to protect the confidentiality of their data, as in military applications [98, 91, 92, 43, 117].

Discretionary access control In contrast to MAC, DAC gives the owner or the creator of an object the freedom to specify who can access the object and what operations can be performed on the object. For instance, if a subject creates a file on his system and wants to share it with other subjects, the subject can control and specify who can access the file. In other words, the access control is based on the discretion or the decision of the owner. DAC is often implemented using access control list (ACL) for objects, where each ACL defines the types of access granted or restricted to individuals or groups of subjects [60]. Unlike MAC, DAC is suitable for applications that do not require the high level of protection that MAC provides and enforces [43, 117]. In DAC, a subject is permitted to access an object based on the identity of the subject and some subjects can also delegate their own access authorities to other subjects. This is regarded as one main difference between DAC and MAC; whereby in DAC a subject with a particular access permission is able to pass that permission to other subjects. However, this has led this model to be complex as well as flexible [92, 91, 60].

In building operating systems, the decision to select MAC or DAC depends on what functionality an operating system intends to provide. For instance, windows-based platforms provide DAC access structures rather than MAC. However, specially developed operating systems such as those created for government agencies and the military, provide a MAC access structure to enforce the level of security needed [43, 117]. Many operating systems such as Linux, Unix and windows NT/SERVER use a DAC access structure [92].

Role-based access control Ferraiolo and Kuhn [35] proposed the RBAC model to overcome the complexity problem associated with the previous two models. In this model, access to objects is restricted based on the business function or role that subjects perform. Unlike DAC, access permissions in RBAC are assigned to roles rather than to subjects' identifiers [98]. In this model, groups of users are created by the administrators who then assign access rights and permissions to the groups; and a user in a group will be able to utilise the access rights and permissions of the group they are placed in [43]. This model is more appropriate for large organisations that are required to change the access rights and permissions more often. This because of the fact that this model allows the administrators to add a subject, an object, or can change access rights and permissions very easily by altering centralised roles without having to manipulate any subject or object in the system. For instance, in a company the administrator can add a new employee to a role rather than creating access rights and permissions for every person who joins the company [43, 117].

All the three models described above, can control access to objects in closed systems but cannot control access to objects in open distributed systems. Salim et al. [98] indicate that the failure of MAC, DAC, and RBAC models to control access to objects in open distributed systems stems from several factors. Firstly, these models require that subjects and objects must already be known before access is granted. Secondly, these models rely on an access control list (ACL) to express policies which are usually stored in a central server under the control of a trusted administrator. Thirdly, users in distributed systems need their rights to be delegated to other users in order for tasks to be shared and accomplished. Finally, the trustworthiness of clients' software/hardware in traditional access control models is questionable.

In order to solve these problems, many studies have been carried out and have resulted in the emergence of newer access control models which are trust management and DRM models. These models are discussed in the following sections.

2.4.6 Trust management model

Generally, this model is intended to solve the problem of traditional access control models that require subjects and objects to be previously known and before access is granted. Hence, this model allows administrators to authorise previously unknown users by the use of PKI and credential based systems [98]. The main difference between traditional access control models and the trust management model is that traditional access control models are centralised and operate under a closed system, where all the parties are known; whereas trust management systems operate in open distributed systems where some parties could be unknown[128]. In the trust management model, each subject is bound to authorisations referred to as credentials which help in determining and judging the capabilities of subjects based on the relevance of subjects' credentials to the local policy of objects' provider. Each subject in the trust management model can be an authoriser, a credential issuer, or a requester [98].

Although the trust management model addresses the problem of dealing with previously unknown subjects in distributed systems, it fails to control access to objects that are sent from the authoriser domain to the requester domain. This is due to the fact that the trust management model, like other traditional access control models, operates on objects within server systems and does not control access to objects that are locally stored at the client-side. Therefore, studies in solving this problem have led to the emergence of the DRM model.

2.4.7 Digital Rights Management (DRM) model

All the access control models that are mentioned so far focus on controlling access to objects within a defined boundary that is either a system or an organisation. However, the DRM model focuses on controlling access to objects regardless of their location which means across systems and organisations. In other words, the DRM model provides an access control mechanism for objects that are already sent from the authoriser domain to the requester domain and which are locally stored in the requester's machine. DRM is commercially-oriented as the authorisation process is based on payment, in the sense that a subject will be authorised to use an object if the subject has paid for it, otherwise the authorisation will be denied. In fact, DRM has emerged to eliminate copyright infringement that caused a huge revenue loss to the owners of copyrighted content.

Liu et al. [64] defined DRM as “A system to protect high-value digital assets and control the distribution and usage of those digital assets”. Harinarayana et al. [42] state that DRM refers to the “technologies and processes that are applied to describe the digital content and to identify the user. Further it refers to the application and enforcement of usage rules in a secure manner”. Hence, DRM may look similar to traditional access control. However, Safavi-Naini and Sheppard [97] assert that unlike traditional access control, DRM protects the content beyond the boundary of systems that controlled by the content owner. Thus, they defined DRM as “persistent access control” to distinguish it from traditional access control models which are unable to prevent users from conforming to any particular usage policy once they have gained access to the content.

DRM systems are a promising solution to prevent copyright infringement, they allow an owner of a digital content to choose who will be consuming the content and how the content will be consumed. More importantly, DRM systems have made the owners of digital content able to control their content in such a way that is impossible to do in physical contents. For instance, an owner of a digital music file can use many restrictions over the music file such as a number of times the music can be played, include an expiry date in which the music file will not operate when it reaches a particular time, prevent copying or allow copying but for a limited number etc. These controls have impressed content owners and introduced a wide range of new business models such as pay-per-download, subscription, pay-per-play, try-before-you-buy and rental.

A DRM protected content is useless by itself. In order to make use of it, consumers must obtain a licence that makes the content operable. Safavi-Naini and Sheppard [97] state that DRM systems associate a content with a license which sets out all rights that is granted to a user by the content owner. The licenses are in a machine-readable and machine-enforceable fashion. The user can only access the content by using hardware and software which are trusted to the content owner, and which will only allow the user to make use of the content according to the rights granted by a license.

In spite of the fact that all DRM systems rely on the approach of associating a license with each content, they have different architectures [42, 9, 86]. Liu et al. [64] point out that although each DRM vendor has different DRM implementation, names and ways to specify the content usage rules from one another, the basic DRM process is the same which often composed of four parties: the content provider, the distributor, the clearing-house (license issuer) and the consumer (see Figure 2.2). Safavi-Naini and Sheppard [97] illustrate DRM architecture as follows: a provider creates content and then sends it to a user in an encrypted form via some distribution channels. The user obtains a license from a license issuer to be able to access the protected content. Licenses are written by Right Expression Language that is a machine-readable and it is used to set out the terms of use of the content and the information required to access the protected content. One of the most important security requirements for a DRM system is that the hardware and the software which the user utilise to access protected content must be guaranteed by its manufacturer to behave in accordance with licenses. However, consumer's device is not trustworthy and this is the major problem of DRM systems and it is apparent since the aim of the DRM system is to prevent consumers from violating copyrighted contents [112]. As a result, each DRM system uses its own proprietary player applications to protect digital contents, which leads to the problem of interoperability.

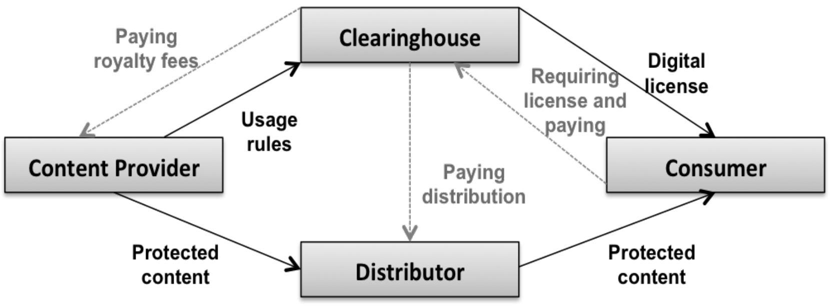

Figure 2.2: The common DRM architecture [adopted from 64]

It is worth taking into consideration that each of the access control models mentioned earlier has focused on their targeted issues. For instance, traditional access control models have focused on controlling access to objects within a closed system that knows the identity and the attributes of the users or processes in advance. The trust management model has also focused on controlling access to objects within a closed system, but it authorises unknown users based on their capabilities and properties. DRM models have focused on controlling access to and usage of objects even after the objects are disseminated. However, as the DRM model is a promising solution for commercial industry, all current DRM systems focus on controlling payment-based dissemination.

As a result, Park and Sandhu [84] defined a model called *Usage Control* that encompasses traditional access control models, the trust management model and DRM model and goes beyond in its definition and scope. "Usage Control (UCON) is a conceptual framework that covers these areas in a systematic manner to provide a general-purpose, unified framework for protecting digital resources" [85, 83]. In UCON, subjects and objects are associated with attributes which can be updated as a result of subjects' actions on objects. Examples of a subject's attributes are user identity, security clearance or role, whilst object's attributes are security labels, owner id, classification or cost. Subjects' and objects' attributes can be updated before usage starts (pre-update), during the usage (on-update), and after the usage is terminated or usage right is revoked (post-update). Access decisions can be evaluated before the requested right is exercised or continuously while the usage right is being exercised. The ability of updating attributes at different stages and evaluating access decisions before or during exercising the usage right make UCON model more expressive than other access control models. However, the UCON model is a conceptional general purpose model that provides no explicit enforcement mechanisms.

The main problem with access control in general is that access rights of programs are only verified at the point of access. At the access point, a program might be denied or granted access to information. Once a program is granted access to information, no further steps are taken to ensure that this program which is given access to information is going to handle the accessed information correctly and securely. Unlike access control, information flow control ensures that a program which is given access to information is going to handle the accessed information securely by tracking how information propagates through the program during execution. In the next section we give an overview of information flow control and focus on type-based approach to information flow control.

2.5 Information flow control

The most widely used technique to prevent information leakage is access control such as Discretionary Access Control (DAC) [60, 44] and Role-based Access Control (RBAC) [100]. Although access control is useful to specify who can access which information, it cannot protect sensitive information against legitimate users. Access control is concerned with the release of information but not its propagation. It provides a guarantee that information is released only to authorised users. However, once information is released to authorised users, it might be leaked maliciously or accidentally to unauthorised users without any further control.

Information flow control is a promising complementary approach to access control to prevent information leakage. It tracks how information propagates through a program during execution to ensure the program does not leak sensitive information. There are various language-based techniques to enforce information flow control statically or dynamically. The former analyses information flow within a program prior to execution while the latter analyses information during execution. Each type of analysis has its own strengths and weaknesses. For example, dynamic analysis has the benefit of permissiveness but incurs run-time overheads, while static analysis has the benefit of reducing run-time overheads but might reject programs that are safe. Detailed discussion about the strength and weakness of both approaches, and suggestions for a hybrid approach can be found in [33]. This thesis focuses on static analysis for secure information flow by the use of type systems. Therefore, the remaining part of this chapter briefly reviews various type systems that are developed to statically analyse information flow in programs. Such type systems are well-known as security type systems which enforce information flow policies. A comprehensive survey of the large body of work on language-based information flow control, can be found in [95].

2.5.1 Security type systems

Denning [29] pioneered the use of static analysis to identify if the information flow of a program satisfies an application-specific confidentiality policy. Following their work, many security type systems have been developed [1, 52, 139] beginning with Volpano et al. [125] and Volpano and Smith [126] who were the first to formulate Denning's secure information flow analysis [28, 29] as a type system and prove its soundness. The intuition is that secure information flow is guaranteed for a program if the program is type-checked correctly.

In security type systems each variable in a program is associated with a security level that represents a flow policy on the use of the value stored in the variable. The security levels associated with programs' variables form a lattice structure, ordered by $\leq$, following an early influential work by Denning [28] who proposed a lattice model of secure information flow. In the lattice model, an information flow policy is defined by a lattice (SC,$\leq$) where SC is a finite set of security classes partially ordered by $\leq$. For example, security classes for confidentiality can be *low* and *high* where *low* $\leq$ *high*, and for integrity can be *trusted* and *untrusted* where *trusted* $\leq$ *untrusted*. Information is allowed only to flow upwards in the lattice. That is information flow from variable x to variable y is allowed if $l_x \leq l_y$. Security type systems enforce such conditions through type checking where the compiler type-checks a program, which contains variables associated with security levels, before execution and ensures that the type-checked program will not violate the information flow policy at run-time.

Information flow in a program might be explicit or implicit [28, 29]. Explicit flow results from assignment operations that assign a variable to another variable. For example, the assignment statement ($x =: y$) contains an explicit flow of information from y to x. On the other hand, implicit flow results from the control structure of a program. For example, the following if-statement (if $x = 0$ then $y := 0$ else $y := 1$) contains an implicit flow from x to y, since after executing the statement the value of y determines whether the value of x is 0 or not. Other kinds of information flows might arise in a program through covert channels [61] such as termination channels, timing channels, probabilistic channels, resource exhaustion channels, and power channels [95]. Such channels are quite difficult to secure since it requires much knowledge of the underlying system and hardware.

A major advantage of security type systems, and static analysis in general, is that it not only controls explicit flows, but also controls implicit flows precisely in all possible execution paths including paths that are not taken at run-time. The typing rules of security type systems control explicit and implicit flows as follows. To control explicit flows as in the example above, the typing rule for assignments requires that $l_y \sqsubseteq l_x$, which means that the security level of variable x must be at least as restrictive as the security level of variable y. To control implicit flows, *program-counter*, written as *pc*, which tracks the security levels of control flow paths is introduced. In the implicit flow example above, the branch taken depends on the value of x, therefore, the *pc* in the then and else clauses will be joined with l_x, written $pc \sqcup l_x$, and the assignment to y is only allowed if $pc \sqsubseteq l_y$.

The majority of security type systems focus on enforcing a property known as *non-interference* [37, 125, 95, 66]. Non-interference for confidentiality requires that public output is independent from secret input, and for integrity requires that trusted output is independent from untrusted input. Various flavours of non-interference exist in the literature to deal with the different powers an attacker might have such as termination-insensitive non-interference, termination-sensitive non-interference, progress-insensitive non-interference, and progress-sensitive non-interference [47]. However, non-interference is a very restrictive property that is hard to meet in practice. This is because non-interference does not allow downgrading of security levels from high to low. In fact, declassifying security levels is needed in many applications. Consequently, various approaches to declassification of information are investigated in the literature. These approaches are surveyed in [96], based on what, where, when, and by whom information can be released. Furthermore, enforcing non-interference can only control how information flows from one security level to another; but cannot control how information at a particular security level is manipulated [63, 13]. For example, regardless of the security level assigned to a variable, the variable can be read, concatenated with itself and saved back as long as these operations only manipulate the variable at the same security level assigned to it. An alternative notion to non-interference based on a security error that can be enforced by a type system as a safety property is proposed in [17].

Broadly, two kinds of information flow policies can be enforced, based on whether the type system is flow-insensitive or flow-sensitive. In flow-insensitive type systems, such as in [125], variables are assigned fixed security levels. Information can flow from variable y to variable x if and only if $l_y \sqsubseteq l_x$, that is the security level of x is at least as restrictive as the security level of y. Illegal implicit flows are avoided by the use of pc as described above, such that assignments to x that occur in loops and conditional branches is allowed if and only if $pc \sqsubseteq l_x$. On the other hand, in flow-sensitive type systems [52], information can flow from variable y to variable x without the restriction $l_y \sqsubseteq l_x$. However, the security level of x must be changed to be the same as the security level of y after the flow of information. The lattice structure of security types in flow-sensitive type systems is used to avoid illegal implicit information flow. For example, an assignment from y to x that occurs in loops and conditional branches must cause the security level of x to be changed to $pc \sqcup l_y$. The flexibility of allowing variables to change their security levels at different points of the program, makes flow-sensitive type systems more permissive, yet secure, than flow-insensitive type systems. This is because flow-sensitive type systems accept more programs, that otherwise would be rejected by flow-insensitive type systems, without jeopardising security.

Myers and Liskov [75, 76, 74, 77] developed a decentralised model for information flow known as *the decentralised label model* (DLM) which was implemented as the language JFlow [73]. Programs written in JFlow can be type-checked statically by its complier to eliminate illegal information flow. The model improves on earlier approaches to controlling the flow of information. The DLM allows users to control the flow of their information by defining their own security policies. In DLM, information is owned by, updated by, and released to *principals* who are the users of the system. The security policies of principals are expressed in *labels*. Each label consists of a set of components that express the security policies by various principals. Each component has two parts, an owner and a set of readers, and is written in the form *owner: readers*. The readers of a component are the principals who this component permits to read the data. Thus, the owner is a source of data whereas the readers are possible destinations for the data. An example of a label is $l = \{o_1 : r_1, r_2; o_2 : r_2, r_3\}$. Here, o_1, o_2, r_1, r_2 denote principals. Semicolons separate two policies (components) within the label l. The owners of these policies are o_1 and o_2, and the reader sets of the policies are $\{r_1, r_2\}$ and $\{r_2, r_3\}$, respectively. However, only r_2 can read the data because it is allowed by both policies of the label l. Therefore, information can be released to a destination if every policy in the label of the information allows the information to be released to the destination. That is all policies in the label must agree to release the information to that destination.

In DLM every variable has a label, when a value is read from a variable it acquires the label of the variable. However, when a value is stored into a variable, the label of the value is forgotten and it acquires the label of that variable. Therefore, the assignment of a value to a variable results in a relabeling of the copy of the value that is assigned. DLM allows relabeling if it is a restriction, that is the new label must remove readers, add owners, or both. More formally, a relabeling from l_1 to l_2 is a restriction, written as $l_1 \sqsubseteq l_2$ if and only if:

$$owners(l_1) \subseteq owners(l_2) \wedge \forall O \in owners(l_1), readers(l_1, O) \supseteq readers(l_2, O)$$

which means that all the policies in label l_1 are guaranteed to be enforced in label l_2. Based on this, the following relabelings are restrictions:

$$\{A : B, C\} \sqsubseteq \{A : B\}$$
$$\{A : B\} \sqsubseteq \{A :; D : E\}$$
$$\{A : B, C\} \sqsubseteq \{A : B; A : C\}$$

Another kind of relabeling can be performed through *declassification* which relaxes overly restrictive policies. DLM allows declassification only by a process which is authorised to act on behalf of a principal whose policy is to be relaxed. Since principals can only relax their own policies, other policies owned by other principals in a label will be safe. Furthermore, DLM enforces all the policies of derived values that occur during computation. For example, when combining two values labeled l_1 and l_2, respectively, the result must have the least restrictive label that maintains all the flow restrictions specified by l_1 and l_2. Since a label is simply a set of policies, the least restrictive set of policies that enforces all the policies in l_1 and l_2 is simply the union of the two sets of policies. This least restrictive label is the least upper bound or join of l_1 and l_2, written as $l_1 \sqcup l_2$. For example, the join of the labels: $\{A : B\}$ and $\{C : A\}$ is $\{A : B; C : A\}$. More details about DLM can be found in [74].

Chothia et al. [23] introduced the *Key-Based Decentralised Label Model* (KDLM) for distributed access control that combines a weak notion of information flow control with cryptographic operations. They developed a type system to enforce access control in a distributed environment while allowing applications to secure themselves by the use of cryptographic techniques. KDLM is different from DLM in that it includes the notion of key names. Unlike DLM where policies are enforced based on principals, KDLM enforces policies based on key names. To reflect the notion of key names, the label format in KDLM is different from DLM. Their approach is motivated by linking type-based approach for confidentiality and integrity of information to the safe use of cryptographic operations.

The basic idea of KDLM is the addition of key names to the type system. Key names are associated with types, similar to labels in DLM, that identify owners and sets of principals that can access protected data. A key name might be either for encryption or for signing. An example of encryption key name K that is generated by the principal P and is accessible to principals $P_1 \ldots P_m$ has the kind: $K : EKey_F(P : P_1 \ldots P_m)$. Each key name is associated with public-private pair of cryptographic keys. The type of the key name constrains which principals can access the private key for that key name, whereas the private key in turn has a secrecy label that cannot allow access to any principals outside those listed in the key name's ACL. It is assumed that each encryption or signing key has exactly one other corresponding key that is used for decryption or authentication, respectively. The notations a^+ and a^- are used to denote the public and private parts of such a key pair, respectively. Then, for the encryption key name K above, we have the typings:

$$a^+ : [EncKey(K)]^{L_1, L_1'}, a^- : [DecKey(K)]^{L_2, L_2'}$$

The kind of the key name K enforces the restriction that the secrecy label L_2 of the private key a^- cannot allow any principal outside of $P_1 \ldots P_m$ to access the key. The authors combined two ideas which are the notion of type-based cryptographic operations to statically check some properties of those operation, and the notion of decentralised labels that combine access control and some form of information flow control. However, their interest was in access control aspect of decentralised labels.

Jeffrey and Zdancewic [122] used a variant of DLM and developed the language SImp with primitive to enforce information flow policies with cryptographic operations. They demonstrated that programs written in their language satisfy the standard non-interference property. Their goal was to incorporate cryptographic operations with language-based information flow security. They asserted that little efforts has been made to develop a theory to incorporate cryptography and information flow mechanisms. They pointed out that it is essential to understand the relationship between cryptography and information flow, particularly when protected data must leave the managed environment provided by the language runtime.

The authors designed a programming language that meets the following three goals. Firstly, the model of the programming language must have suitable abstractions for enforcing information flow policies specified in security labels cryptographically. Secondly, the language should free the programmer from the burden of manually managing keys and information flow policy labels. Thirdly, programs written in the language should be proved to satisfy the standard noninterference properties.

Abadi [1] developed a type system for concurrent language, the spi calculus, to protect the secrecy of data in security protocols. In his approach, each data and channel is associated with a label which can be secret or public. Data associated with a secret label should not be transmitted on channels that are associated with a public label; and channels associated with a secret label should not be made available indiscriminately. The type system provides a guarantee that secret inputs will not be leaked if the protocol is type-checked.

The author mentioned that principles and rules developed in his work is not necessary since, like most practical static typechecking disciplines, they are incomplete. Also, they are not sufficient since they only focus on secrecy and ignore all other security issues. However, they provide useful guidelines, and the typing rules are tractable and precise which allow him study them in detail and to prove secrecy properties. In such a way the author was able to establish the correctness of the informal principles within a formal model.

Chaudhuri and Abadi [21] developed a type system for a pi calculus with file system constructs to check access control and limit the dissemination of file names and content in a fairly standard file system. They associate types with file names and with groups of clients which represent the reach of the type. The reach of the type is the group of clients that is allowed to share file names and contents among themselves. The type system guarantees that file names and contents will not be leaked to anyone outside the reach of their types.

For example, assume there is a client C_1 who creates a secret m that must not be shared with anyone. Then, if C_1 writes m to a public file and another client C_2 attempts to read this public file, then m would not be secret to C_1. This is because the public file will contain the secret m. The authors approach is to analyse such a system and it will only typecheck if C_2 does not have read access to that file. More interestingly, it is possible for such system to typecheck if C_2 does not attempt to read the public file, even if C_2 has read access. The authors illustrate various examples which indicate that their type system is fairly permissive. However, their type system will fail to typecheck any process that violates secrecy intentions.

Takeuchi et al. [119] and Honda et al. [48] were the first to propose a formalism to structure interaction and statically analyse communication protocols known as session types. In session types, communication protocols are expressed as types to specify the topic of conversation, the sequence, and the direction of the communicated messages. The type system then statically analyses whether agents exchanging messages observe the correct protocols or not.

Vasconcelos [121] presented an example of how session types can specify the interaction in simplified distributed auction system with three players who are sellers that want to sell items, auctioneers that sell items on their behalf, and bidders that bid for an item being auctioned. The protocol for sellers is as follows: sellers can invoke only one operation on an auctioneer which is selling. In invoking this operation, they must provide the auctioneer with a description of the item to be sold (a string), and the minimum price they are willing to sell the item for. This protocol can be specified as follows, where $\oplus$ denotes the choice available to sellers, and ! denotes the output of a value.

$$\oplus\{selling : !String.!Price \ldots\}$$

Now, sellers should wait the outcome of their request, where two things can happen. Firstly, the item was sold, and secondly, the item was not sold. The protocol continues as follows, where & denotes the range of alternatives offered by the seller at this point, and ? denotes input.

$$\&\{sold : ?Price \ldots, notSold : \ldots\}$$

In both cases the protocols should halt. This is indicated by the mark **end**. The complete protocol as seen by the seller can be concisely described.

$$\oplus\{selling : !String.!Price. \& \{sold : ?Price.end, notSold : end\}\}$$

The protocol for auctioneers is as follows: we know that auctioneers must offer a selling alternative, and if such alternative is taken, then they must accept a string (the item be sold) followed by the price the seller is asking.

$$\&\{selling : ?String.?Price \ldots\}$$

The auctioneer then puts the item on sale, and gets back to the seller with one of the possible outcomes: *sold* or *notSold*.

$$\oplus\{sold : !Price \ldots, notSold : \ldots\}$$

Putting everything together we have two session types, the first for the seller, the second for the auctioneer.

$$\oplus\{selling : !String.!Price. \& \{sold : ?Price.end, notSold : end\}\}$$

$$\&\{selling : ?String.?Price. \oplus \{sold : !Price.end, notSold : end\}\}$$

The above description leads to safe interaction between sellers and auctioneers. It is clear by the session types of the two partners that when seller selects the *selling* choice, the auctioneer offers that exact choice, and conversely for choices *sold* and *notSold*. Furthermore, when the seller outputs a value, the auctioneer inputs a value of the same type, and when the seller ends the protocol, so does the auctioneer. Such two session types is said to be dual, a notion central to session types. A large body of work on session types followed [119, 48]. For a good overview on session types and a survey of recent work we refer to [30].

There are many security type systems exist in the literature to control the access to and flow of information in programs, or to analyse security protocols. They are applied in different problem domains to ensure different security properties in programs. Our aim is not extend these type systems, but rather is to apply such static analysis technique to our problem domain. In particular, we apply such static analysis technique to analyse commands manipulating files in a Unix-like file system. We statically analyse these commands before execution to protect shared files against possible misuse. Misuse of shared files occur by commands that violate files policies. The novelty of our approach is to use a type system which is a static analysis technique in a highly dynamic environment which is a file system. Files policies are not static and they might change overtime. The dynamic nature of files policies should be considered by the type system to prevent any possible misuse.

Our type system enforces access control and information flow requirements. It enforces access control by restricting commands to be issued on files based on files permissions. That is, a command can be issued on a file only if the permission of that file allow such command to be issued. It enforces information flow by restricting information in source files to flow to destination files if and only if the permissions of the destination files are the same or more restrictive than the permissions of the source files. In this way, we can ensure that access control requirements are not violated by information flow between files.

In Chapter 5 we present our type system which can be thought of as a reference monitor that check commands before execution, and only allow those commands which do not cause misuse of files. We postpone discussion about our approach to the end of Chapter 6, once concepts and techniques used in our approach have been clarified.

2.6 Summary

In this chapter, we provided the necessary background and related work for topics discussed throughout the thesis. We defined the activity of file sharing and showed the evolving history of file sharing methods. Previous studies that investigate people's practices of the activity of file sharing were reviewed, and we showed that while previous studies provided valuable answers to fundamental questions that could lead to better design of file sharing methods and access control models, they ignore a significant question, namely how files can be propagated and accessed after their propagation. Characterising the activity of file sharing based on how files can be propagated and accessed leads to better understanding of how the activity of file sharing is performed. From such characterisation, various classes of the activities of file sharing can be deduced, which can be thought of as policies for the sharing activities. A protection mechanism enforces these policies, therefore, will not only protect the shared files but also allows users to share their files as desired. In Chapter 4 we provide a characterisation of the activity of file sharing based on these two factors.

Policies can be enforced to counter various kinds of attack. We divided these attacks into communication, perimeter and insider attacks. We defined each kind of attack and reviewed the protection mechanisms developed to counter each of them. Our interest is to protect the shared files from authorised users; therefore, the literature on insider threats is the most relevant to our work. We showed that the literature on insider threats is lacking a clear definition of what an insider is and of insider threats. This has complicated the problem to be solved; and we believe that the slow progress in the field to counter the insider threat is caused by the single definitions of the insider problem and its threats, as attempted in previous work. We suggest that the insider problem should be divided into smaller problems that can be defined, studied and solved independently. In Chapter 3 we propose a classification of the insider threat problem; focus on one category that is related to file sharing and define the threats imposed by insiders in this category. It is threats such as these that we are concerned to prevent in this thesis. Since the threats are precisely defined, we follow a prevention approach rather than a detection approach. Generally, the prevention approach falls into two forms, namely: access control and information flow control. We reviewed the literature on these approaches and focused on a type-based approach for information flow control; which is the approach that we adopt to tackle our particular class of insider problem in Chapter 5.

Chapter 3

The Insider threat problem

This chapter proposes an approach to classify the insider problem, and provides precise definitions of the insider and the insider problem. Based on the proposed classification, it defines and focuses on one class of the insider problem that is related to file sharing.

3.1 Introduction

Protecting the shared files from the perspective of insider security is a challenging problem. It has always been recognised that preventing policy violation by authorised users is more challenging than those who are not. Authorised users have access privileges that make it hard to prevent or detect policy violation. Providing a mechanism to protect the shared files from insiders requires an investigation into two fundamental questions, which we address in this chapter.

- Firstly: What is the insider problem?

The problem with the insider security literature is that there is no widely accepted definition of what is an insider; and there is no clear distinction between insiders and outsiders. Who is considered an insider by someone might be an outsider for someone else. Therefore, protecting the shared files from insiders without knowing who constitutes that insider is meaningless. By surveying the previous work on insider security, we argue that the insider problem is significant and that no single definition can encompass the problem as a whole, which is what most researchers have attempted to do. Researchers approach the problem of insider security by defining two terms which are *insider* and *insider threats.* In the literature, insiders have always been defined and differentiated from outsiders by either being inside the network perimeter, trusted, authorised, or knowledgeable about the information system, or possibly all of these. Definitions based on these factors are either ambiguous or insufficient. For instance, definitions based on trust exclude those untrusted insiders who might be authorised to access an organisation's assets; definitions based on the network perimeter exclude those outsourced organisations and contractors who might be authorised to access the internal network remotely; definitions based on authorisation exclude those who illegitimately acquire authorisation credentials in order to access an organisation's assets as if they were authorised insiders and definitions based on knowledge include previous insiders who are no longer working for the organisation. Defining the insider is not as useful as defining the threats that an insider can pose for an organisation. Definitions of *insider* in the literature attempt to differentiate insider attacks from outsider attacks. However, such differentiation cannot be recognised by the trust, knowledge or authorisation that the insider might have but rather by the types of attacks and misuse of a particular asset of an organisation.

Definitions of *insider threat* in the literature have always relied on the definition of the insider; as the insider threat is seen as the damage caused to an organisation by an insider. However, the insider problem is huge and defining the insider threat based on all possible attacks or misuse that insiders might perform in an organisation is rather complicated and ambiguous. This approach to defining the insider problem based on a strict definition of the insider and a broad definition of the insider threat is only helpful to get an idea about the field, but will never help to solve the insider problem.

To make progress in the field and find a solution to the insider problem, we suggest that the problem should be classified into several categories which can be defined, studied and solved independently and which later can be combined to solve the problem as a whole. The authors in [89, 51, 90] identified different factors for an insider that can be used for defining a taxonomy of insider threats. One of these factors is the distinction between the acts of masqueraders (e.g, an individual with a stolen password), traitors (malicious legitimate users) and naive or accidental use that results in harm. Although such distinction is useful in classifying types of insider based on their intentionality or characteristics, it does not help in classifying the insider problem or differentiating insider from outsider attacks. However, such classification can be of great value if different types of attacks and misuse are associated with each type of insider.

We believe that the classification of insider attacks by Silowash et al. [107] is the first step towards a useful study of the insider problem. However, such classifications are rather general and should be further classified into more details. For instance, IT sabotage can be performed by an insider who initiates a Denial of Services Attack, deleting critical data from an application he is authorised to use by his machine, planting a logical bomb in software that other employees are using to delete data or making the software inoperable at critical times etc. Also, theft of IP can be performed by an insider who accesses a database in a server illegitimately in order to download IP or sensitive files to his machine; writes down or memorises customers' information that is rendered by an application which the insider is using; or it can be the result of sharing IP and sensitive files with the insider legitimately. Therefore, each class of attack indicated by Silowash et al. [107] should be further classified into different categories that can be studied independently because it is impossible to provide a single solution for all insider IT sabotage attacks.

- Secondly: What is the insider misuse?

Defining the insider problem and the insider precisely is the first step towards protecting shared files from insiders. What is more important is identifying the misuse that can be performed by insiders. Misuse is any action taken by the insider that violates the confidentiality, integrity or availability of a particular asset. By knowing the misuse that the insider can perform on the shared files, we can derive the different types of protection that are required to protect those shared files.

The rest of this chapter is organised as follows: in Section 3.2 we propose an approach to classifying the insider problem, provide a precise definition of the insider and the insider problem, and identify the class of insider problem we are concerned with in this thesis. In Section 3.3 we investigate the different types of misuse that give rise to our class of insider problem and characterise the protection requirements against them. We precisely define the class of insider problem that we tackle throughout this thesis. In Section 3.4 we summarise this chapter.

3.2 Classifying the insider threat problem

There are three factors which play an important role in classifying the insider problem which are: the type of activity that deals with an asset in an organisation; the type of asset that needs to be protected; and the type of attack that targets the asset. Figure 3.1 illustrates how each factor helps us to classify the insider problem.

The activity: Activities are identified by the organisation for its partners, contractors, and employees to perform a particular job, and might be different from one organisation to another. Each activity will differentiate insiders from outsiders, as an insider will be a

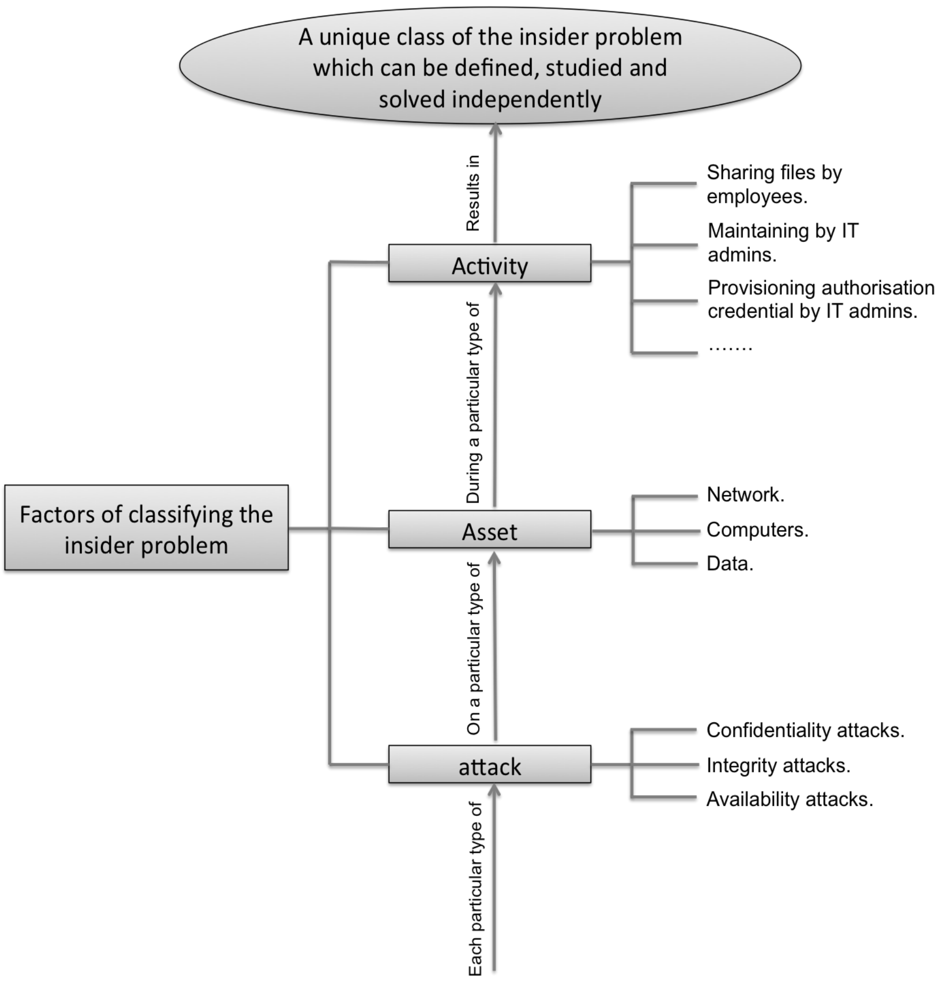
A unique class of the insider problem which can be defined, studied and solved independently
Results in
Activity
Sharing files by employees.
Maintaining by IT admins.
Provisioning authorisation credential by IT admins.
…….
During a particular type of
Factors of classifying the insider problem
Asset
Network.
Computers.
Data.
On a particular type of
attack
Confidentiality attacks.
Integrity attacks.
Availability attacks.
Each particular type of

Figure 3.1: Classifying the insider problem

person who is legitimately given an activity by an organisation to perform a particular job. Therefore, the activity will lead to identifying who is the insider and what that insider is doing. The type of activity that insiders perform in an organisation are various and organisation-specific. Examples of activities that are given to insiders are file sharing, updating customer information, installing software onto an organisation's devices, setting up an organisation's network or provisioning authorisation credentials for an organisation's employees, etc.

The asset: The assets that need to be protected are identified by an organisation based on a clear description of activities in the organisation, such that each activity will involve one or more assets to be dealt with. For example, if an activity in an organisation is employees sharing files with each other, the asset will be the file being shared, which contains sensitive information. Other examples of activities and assets are: an IT administrator who provisions authorisation credentials for an organisation's employees, where the asset is the authorisation credential; a software developer who writes software scripts for an organisation's computer, where the asset can be the software itself or the computers that run the scripts; or, a network administrator who sets up the organisation's network and maintains it, where the asset is the network. Generally, the assets can be of three types which are the network which connects devices together, the devices which contain the data, or the data itself.

The attack. The attacks that target the asset can generally be of three types, which are: availability attacks, confidentiality attacks and integrity attacks; each of which can be performed in different ways which might require either physical security or IT security. Choosing which type of attack to prevent is determined by the type of protection required for the chosen asset. For instance, if the asset is the network which needs to be available all the time, availability attacks should be prevented. On the other hand, if the asset is data that needs to be secret, confidentiality attacks should be prevented and so on. Therefore, the asset will determine which type of attack should be prevented.

Based on these three factors, we define the insider and the insider problem precisely as follows.

Definition 3.2.1. An insider is a person who is legitimately given an activity by an organisation that entails dealing with that organisation's assets.

Definition 3.2.2. The insider problem is particular types of attack that are performed by insiders on particular types of assets of an organisation during particular types of activity.

Therefore, we can classify the insider problem into several categories based on these three factors such that *each particular type of attack by insiders on a particular type of asset of an organisation during a particular type of an activity will result in a unique class of the insider problem which can be defined, studied and solved independently.* For example, one class of the insider problem is preventing confidentiality attacks on sensitive files by employees when they share them with one another. Another class might be preventing availability attacks on an organisation's network by IT administrators when they maintain it, or preventing integrity attacks on customers' information by employees when they update them etc.

Our concern in this thesis is not to classify the insider problem thoroughly; rather we have provided an approach for such classification. However, we are interested in one class of insider problem which is related to file sharing. The activity in this class of problem is thus file sharing; the asset is the file being shared; and the attacks we are concerned with are confidentiality and integrity attacks. Since file sharing is not only an activity that is performed by an organisation's employees but also one that can be performed among friends, family members or colleagues; we will look at this class of insider problem from a broader perspective to include any individuals performing such activity. In other words, the insiders in our class will be the recipients, whether they are employees, friends or family members.

3.3 Protecting the shared files

Although we defined our class of insider problem in the previous section, the attacks we are concerned with (i.e. confidentiality and integrity attacks) are still vague. These attacks can be performed in different ways, which in turn require different types of protection. Claiming that a particular protection mechanism can protect the confidentiality of the files is not enough. Instead, one should claim that a particular protection mechanism can protect the confidentiality of the files under specific kinds of attack. Therefore, in order to protect the confidentiality and the integrity of the shared files from insiders (i.e. recipients), the different attacks and misuse that affect the confidentiality and the integrity of shared files must be identified. Generally, protection of the shared files can be realised from two different angles: protecting the shared files while in transit, and protecting the shared files when they are received by the recipients. In this section, we characterise the protection required by the shared files against different types of attack and misuse that can occur during the activity of file sharing.

3.3.1 Protecting the shared files in transit

This type of protection prevents attacks on the file while it is being transferred from the owner to the recipients. We divided these attacks into confidentiality attacks and integrity attacks as follows:

Confidentiality attacks. These attacks lead to the disclosure of the shared files to unauthorised users and can occur in two ways. Firstly, someone eavesdrops or monitors the communication between the owner and the recipient to obtain knowledge about the files. We refer to such an attacker as an *interceptor*. Secondly, someone pretends to be the original recipient in order to deceive the owner and obtain the files. We refer to such an attacker as a *masquerader*. Therefore, there should be two types of protection to prevent unauthorised disclosure of the shared files in transit as follows: Protecting the confidentiality of files from *interceptors*; and protecting the confidentiality of files from *masqueraders*.

Integrity attacks. These attacks lead to unauthorised modification of the shared files by unauthorised users. The attacker in such attacks pretends to be the original owner to deceive the recipient by sending them files as if they came from the original owner. These files can either be entirely new files or a modified version of the original files. We refer to such an attacker as a *masquerader*. Therefore, there should be one type of protection to prevent unauthorised modification of shared files in transit which is protecting the integrity of files from *masqueraders*.

3.3.2 Protecting the shared files at the recipient

This type of protection prevents misuse of the file after it has been received by legitimate recipients. This misuse can affect the confidentiality and integrity of the files; and can be committed by three different entities which are: *malicious* recipients, *naive* recipients or *masqueraders*. Below we define these three entities and describe the differences between them.

Definition 3.3.1. Malicious recipients are untrusted legitimate recipients who deliberately misuse the shared files.

Definition 3.3.2. Naive recipients are trusted legitimate recipients who accidentally misuse the shared files.

Definition 3.3.3. Masqueraders are unauthorised users who claim to be legitimate recipients to acquire their devices which contain the shared files and misuse these files.

The reason behind differentiating these three entities is that files should be protected against each of them differently. For example, protecting the shared files against malicious and naive recipients is different from protecting them against masqueraders. Malicious and naive recipients are legitimate recipients who might or might not be allowed to view or edit the files. However, masqueraders are unauthorised users who must not be allowed to view or edit the files at all. Moreover, protecting the shared files against naive recipients is different from protecting them against malicious recipients. The former are trusted to not manipulate files in an unauthorised manner, while the latter are untrusted, and might strive to circumvent any protection to misuse the files.

It should be noted that by definition masqueraders are not insiders, since they are not legitimate recipients. However, from the system point of view they are considered insiders, since they gain access to the system as if they were legitimate recipients. That is, they claim legitimate recipients identities to deceive the system, and thus the system cannot differentiate them from legitimate recipients (i.e. insiders). However, masqueraders usually present a vague area where insider and external attacks overlap. It is reasonable to classify masqueraders misuse as external attacks, since they are not legitimate recipients or more precisely they are not legitimately given an activity. Similarly, it is reasonable to classify their misuse as insider attacks, since they gain access with identities of legitimate recipients and the system will perceive them as legitimate recipients.

We eliminate this vague area by classifying the attacks of such unauthorised users into attacks they perform to become masqueraders, and attacks they perform when they have become masqueraders. The former attacks are performed to obtain the credentials of legitimate recipients to claim their identities to the system, and thus perform the latter attacks. These attacks are considered external attacks which can be in the form of fishing, social engineering, brute-force, and spoofing attacks, to name a few. Once such attacks are performed successfully, the attacker will become a masquerader and can perform the latter attacks. The latter attacks are performed to misuse the privileges of the claimed identities of legitimate recipients. These attacks are considered insider attacks since the attacker will be recognised by the system as a legitimate recipient with the same privileges as the victim legitimate recipient.

Since we are concerned with insider attacks we focus on the latter kind of attacks, and thus, it is essential to remember that masqueraders have already obtained credentials to access the system as legitimate recipients. We are only interested in whether that credentials are obtained with or without legitimate recipients cooperation. This is because protecting the shared files against masqueraders who are cooperating with legitimate recipients is different from those who are not. Therefore, we differentiate between two kinds of masqueraders, those who obtain credentials with legitimate recipients cooperation, and those who obtain credentials without legitimate recipients cooperation. Based on this, we classify masqueraders misuse into accidental misuse and deliberate misuse. Masqueraders misuse is accidental if credentials are obtained without legitimate users cooperation. For example, an unauthorised user stealing a legitimate user password will result in accidental misuse since such misuse is unintended by the legitimate user. On the other hand, masqueraders misuse is deliberate if credentials are obtained with legitimate users cooperation. For example, a legitimate user passing his passwords directly to an unauthorised user will result in deliberate misuse since such misuse is intended by the legitimate user.

We assume that devices of legitimate recipients have unique identifiers that cannot be forged, and shared files can only be accessed through legitimate recipients devices. Therefore, the only way for unauthorised users to become masqueraders is through physical acquisition of legitimate recipients devices. We divide misuse which can be committed by the three entities defined above into confidentiality misuse and integrity misuse as follows:

Confidentiality misuse. Confidentiality misuse is that which leads to the disclosure of the shared files to unauthorised users, and which can be done in the following way: Firstly, the shared files can be viewed by a legitimate recipient who is not allowed to view the files. The files can be viewed accidentally by a naive recipient or deliberately by a malicious recipient. Secondly, the device of a legitimate recipient which contains the shared file can be acquired by an unauthorised user, which we refer to here as a masquerader, who discloses the shared files. Such acquisition can be either accidental, as when an unauthorised user steals the device of a naive recipient; or deliberate, as when a malicious recipient lends his device to an authorised user. Thirdly, the shared files can be sent from a recipient device through a file sharing method to unauthorised users who view the files. In this case, the file can be redistributed either accidentally by a naive recipient, deliberately by a malicious recipient or accidentally by a masquerader who found a legitimate recipient's device unattended.

In view of this, there should be seven different types of protection to prevent unauthorised disclosure of the shared files by the recipients as follows: Protecting the confidentiality of files from accidental disclosure to a naive recipient; protecting the confidentiality of files from deliberate disclosure to a malicious recipient; protecting the confidentiality of files from accidental redistribution by a naive recipient; protecting the confidentiality of files from accidental redistribution by a masquerader; protecting the confidentiality of files from deliberate redistribution by a malicious recipient; protecting the confidentiality of files from accidental disclosure by a naive recipient to a masquerader; and protecting the confidentiality of files from deliberate disclosure by a malicious recipient to a masquerader. Since the last two types of protection have a similar impact, which is disclosing the file to masqueraders, we refer to them as protecting the confidentiality of files from accidental or deliberate disclosure to a masquerader.

Integrity misuses. Integrity misuses are those misuses which lead to unauthorised modification of the shared files. Such unauthorised modification can be either modifying shared files that do not allow any modification; or modifying shared files that allow partial modification, in an unauthorised manner. In both cases, the file can be modified in three ways. Firstly, the file can be modified accidentally by a naive recipient. Secondly, the file can be modified deliberately by a malicious recipient. Thirdly, the file can be modified accidentally by a masquerader who finds a legitimate recipient's device unattended.

Therefore, there should be three different types of protection to prevent unauthorised modification of the shared files by the recipients as follows. Protecting the integrity of files from accidental modification by a naive recipient; protecting the integrity of files from accidental modification by a masquerader; and protecting the integrity of files from deliberate modification by a malicious recipient.

Below we classify the aforementioned protections into two types; namely, protection of files in transit and protection of the files at the recipients.

Protection of files in transit: This can be further divided into confidentiality protection and integrity protection.

- Confidentiality protection
  - Protecting the confidentiality of files in transit from *interceptors*
  - Protecting the confidentiality of files in transit from *masqueraders*
- Integrity protection
  - Protecting the integrity of files in transit from *masqueraders*

Protection of files at the recipients: This can be further divided into protection against accidental misuse when sharing with trusted recipients and protection against deliberate misuse when sharing with untrusted recipients.

Accidental misuse: this can be further divided into accidental misuse of confidentiality and accidental misuse of integrity.

- Accidental misuse of confidentiality:
  - Protecting the confidentiality of files at the recipients from accidental disclosure to a naive
  - Protecting the confidentiality of files at the recipients from accidental disclosure to a masquerader
  - Protecting the confidentiality of files at the recipients from accidental redistribution by a naive

  - Protecting the confidentiality of files at the recipients from accidental redistribution by a masquerader

- Accidental misuse of integrity:
  - Protecting the integrity of files at the recipients from accidental modification by a naive
  - Protecting the integrity of files at the recipients from accidental modification by a masquerader

Deliberate misuse: this can be further divided into deliberate misuse of confidentiality and deliberate misuse of integrity

- Deliberate misuse of confidentiality:
  - Protecting the confidentiality of files at the recipients from deliberate disclosure to a malicious
  - Protecting the confidentiality of files at the recipients from deliberate disclosure to a masquerader
  - Protecting the confidentiality of files at the recipients from deliberate redistribution by a malicious
- Deliberate misuse of integrity:
  - Protecting the integrity of files at the recipients from deliberate modification by a malicious

Figure 3.2 illustrates 13 types of protections that might be required to protect the files in transit and at recipients. The protection of files in transit is concerned with preventing external attacks, while the protection of files at the recipients is concerned with preventing insider attacks.

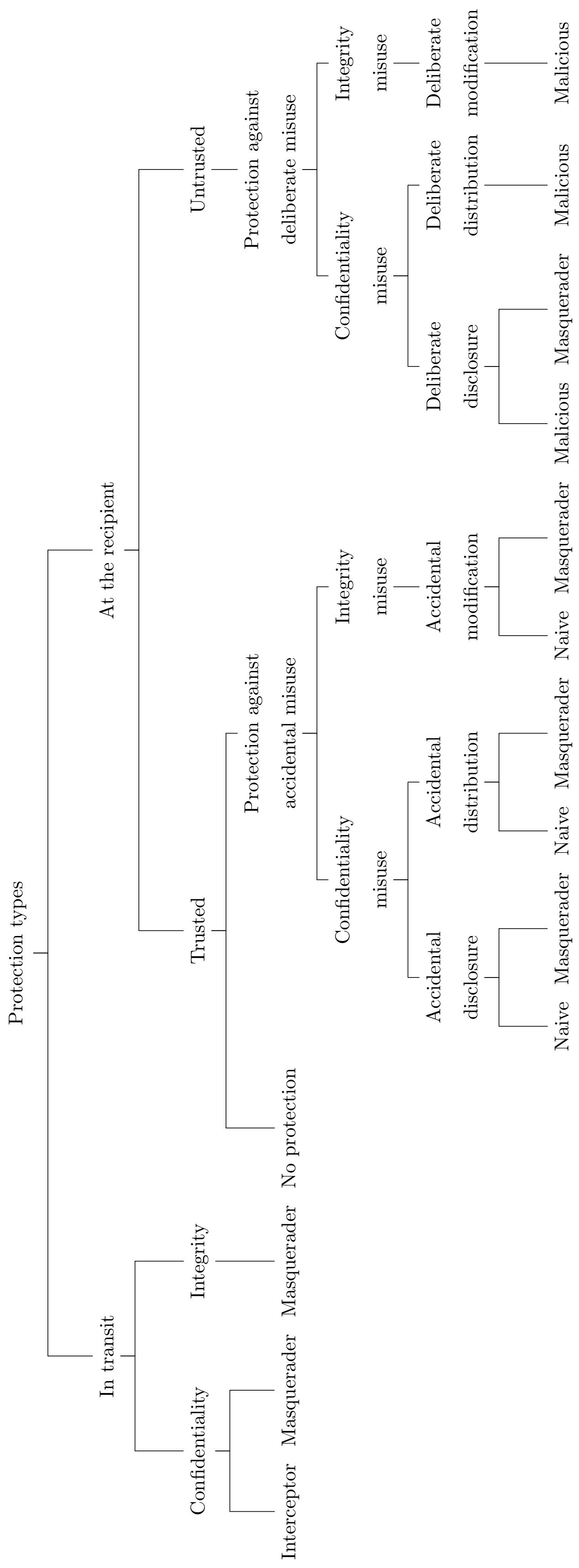

Figure 3.2: Types of protection of the shared files

The characterisation of the protections required by the shared files at recipients, illustrates the different ways files can be misused by different types of insider. This characterisation makes it clear which type of insider misuse needs to be prevented in a particular sharing scenario. For instance, misuse by a masquerader does not need to be prevented if the machine containing the file resides in a locked room which unauthorised users cannot access. Also, deliberate misuse by malicious insiders does not need to be prevented if the file is shared with trusted recipients. A major advantage of this characterisation is the avoidance of the chaos that exists in the literature with respect to distinguishing insider attacks from external attacks, and between types of insider attacks.

The focus of this thesis is on accidental misuse that affects the confidentiality and integrity of sensitive files during the activity of file sharing. Therefore, the class of insider problem which we investigate in this thesis can be defined as follows:

Definition 3.3.4. Our class of insider problem is to protect sensitive files against accidental misuse of confidentiality and integrity by trusted recipients during the activity of file sharing.

Our approach to protect files against accidental misuse is through controlling operations to be performed on them in a Unix-like file system, where users manipulate files by issuing various commands such as `cp`, `mv`, and `cat`. Commands issued to manipulate files are controlled by checking them before execution, and only those commands which do not misuse the files are allowed to be executed. However, commands that misuse files cannot be identified unless there are policies dictate what are and are not allowed to do with the files. For example, such policies can dictate that a command which reads a file can only be issued on a file that can be read, whereas a command which writes into a file can only be issued on a file that can be written into. Then, issuing a read command on a file that is not allowed to be read is consider misuse. Therefore, commands that misuse files are those which violate files policies.

In the next chapter, we characterise the activity of file sharing, and from such characterisation we derive a set of policies that is useful in practice. Such policies should be enforced to prevent accidental misuse of shared files. In Chapter 5, we present our approach to enforce these policies in a Unix-like file system. We focus on enforcing one particular policy and discuss extensions to enforce other policies in Chapter 6.

3.4 Summary

In this chapter we have studied one category of insider threat problem that is concerned with file sharing; in particular, protecting the shared files against insider misuse. We investigated two fundamental questions for the design of a protection mechanism against insider misuse. Since the insider problem is not well-defined in the literature and the insider is not clearly identified, we have proposed a classification of the insider threat problem and defined the insider and the insider threat problem more precisely. Defining the insider problem and identifying the insider precisely are the first steps towards protecting shared files against insiders. More importantly, misuse that insiders might perform on the shared files should be identified. We have looked at different insider misuse of shared files and characterised the protection requirements of the shared files against each of these. The focus of this thesis is on the accidental misuse of shared files; in particular, protecting shared files against accidental disclosure, distribution and modification by recipients.

Chapter 4

Characterising the activity of file sharing

This chapter characterises the activity of file sharing based on how files can be propagated and accessed after their propagation. It defines a framework based on this characterisation that can be used to classify the activity of file sharing and available file sharing methods. It shows how the different classes of the activity of file sharing can be enforced to avoid the different types of accidental misuse identified in the previous chapter.

4.1 Introduction

Although the different types of misuse of the shared files and the protection requirements are identified in the previous chapter, the activity of file sharing is still ambiguous. Some people conceive the activity of file sharing as sending an email attachment, while others conceive of it as making files available to others through peer-to-peer networks. Designing a mechanism that provides the various types of protection without taking into account how the activity of file sharing is performed, is not very useful. This is pointed out by previous studies, which showed that some people might avoid secure methods of file sharing and utilise insecure methods because they were more suitable for the task of sharing, even though security was a concern for them. For instance, employees in organisations might be forced to utilise particular sharing methods because they are secure. However, since these methods have been built with only security in mind, they might not be suitable for the task of sharing that employees need to get their job done. Hence, employees usually tend to utilise other sharing methods that might be insecure to avoid obstacles found in secure methods, and hence, putting the organisation's confidential files at risk. To avoid such issues, the different ways of file sharing should be considered when designing a protection mechanism, so that it will not only protect the shared files, but also allow various sharing tasks to be performed.

The activity of file sharing is performed by individuals for various purposes, be they professional or personal. The purpose of performing the activity of sharing makes it obvious with whom the files are to be shared (e.g. family, friends, colleagues, or anyone); which type of file is to be shared (e.g. music, photo, video, business documents, etc.); and which method of sharing is to be utilised as the most suitable for that sharing purpose (e.g. secure, convenient, available to everyone etc.). These factors are discussed in the literature and summarised in Table 2.3 in Chapter 2.

However, there are two factors that are clearly affected by the purpose of sharing and which have been overlooked by previous studies. These factors are file propagation and access, which can be different according to sharing purpose. To the best of our knowledge, we are not aware of any work that characterises the activity of file sharing based on these two factors. Therefore, in this chapter we characterise the activity of file sharing according to how files can be propagated and how files can be accessed after their propagation. Based on this characterisation, we define a framework to classify the activity of file sharing into different categories. We show that enforcing these categories, which can be thought of as policies, can prevent various forms of accidental misuse of shared files, which are identified in the previous chapter.

The rest of this chapter is organised as follows: in Section 4.2 we present the different ways of how files can be propagated. In Section 4.3 we present the different ways of how files can be accessed after their propagation. In Section 4.4 we define a framework which can be used to classify both the activity of file sharing and the available file sharing methods. We discuss the proposed framework in Section 4.5 and summarise this chapter in Section 4.6.

4.2 How files are propagated

4.2.1 Publish vs. Share:

Files can be propagated in two main ways depending on their sensitivity. Confidential files are only released to selected individuals while non-confidential files are released to everyone. Available file sharing methods can either allow people to share files with selected individuals (suitable for confidential files) or allow people to share files with everyone (suitable for non-confidential files). A few file sharing methods provide both options. We will use the term *share* to refer to a file that is released to selected individuals and the term *publish* to a file that is released to everyone. Publishing or sharing files can be performed in different scenarios.

In general, files can be released either to a person, a group of people or everyone, we refer to them as One, Group, and Many respectively. However, the files received by the recipients who can be One, Group and/or Many, might belong to One, Group or Many. Therefore, including Group as a category of sharing we have 9 different categories that can describe all the possible ways of how files are shared or published and are described below.

1. OneToOne: This describes a situation when a particular owner of files wants to share them with a particular recipient who is known in advance. For example, Alice wants to share her file only with Bob.

2. OneToGroup: This describes a situation when a particular owner of files wants to share them with a set of recipients whose number and identities are known in advance, and who receive the same copies of the shared files. For example, Alice wants to share her file only with her colleagues Bob, Carol, and Dave.

3. OneToMany: This describes a situation when a particular owner of files wants to share them with a set of recipients whose number and identities are not known, and who receive identical copies of the shared files. For example, Alice wants to share her file with everyone on the Internet, regardless of who they are.

4. Group: This describes a situation when owners of files whose number and identities are known in advance want to share their files with each other. For example, Alice, Bob, and Carol want to share their files only with each other.

5. GroupToOne: This describes a situation when a set of owners of files whose number and identities are known in advance, and who share their files with each other, want to share these with a particular recipient who is known in advance. For example, Alice, Bob and Carol who are sharing their files with each other want to share them only with their colleague Dave.

6. GroupToGroup: This describes a situation when a set of owners of files whose number and identities are known in advance and who share their files with each other, want to share them with a set of recipients whose number and identities are known in

advance, and who receive identical copies of the shared files. For example, Alice, Bob and Carol who are sharing their files with each other want to share them only with colleagues in the same department.

7. GroupToMany: This describes a situation when a set of owners of files whose number and identities are known in advance and who share their files with each other, want to share them with a set of recipients whose number and identities are not known and who receive identical copies of the shared files. For example, Alice, Bob and Carol who are sharing their files with each other want to share them with everyone on the Internet, regardless of who they are.

8. ManyToOne: This describes a situation when a set of owners of files whose number and identities are not known in advance and who do not share their files with each other, want to share them with a particular recipient who is known in advance. For example, applicants for a particular job want to share their document files only with Alice, who is the employer.

9. ManyToGroup: This describes a situation when a set of owners of files whose number and identities are not known in advance, and who do not share their files with each other, want to share them with a set of recipients whose number and identities are known in advance, and who receive identical copies of the shared files. For example, applicants for a particular job want to share their document files only with Alice, Bob and Carol, who are the employees responsible for recruiting new staff.

Figure 4.1 illustrates these categories and classifies them into either publish or share. Note that we exclude one situation that does not make sense which is $M' \rightarrow M$, since any of the owners can be one of the recipients and vice versa.

4.2.2 Static vs. Dynamic vs. Transfer mode

In any of the categories of file propagation described above, files can be moved from an owner to a recipient in different ways. For instance, the original file can be moved physically as an object in the real world, leaving no copies behind; or a copy of the original file can be moved to the recipient. In the latter case, the moved copy can be either dynamic or static. Below we describe each one of them.

Publishing or sharing in Static Mode: This describes a scenario where independent copies of the original file are moved from the owner to the recipients. Any changes made

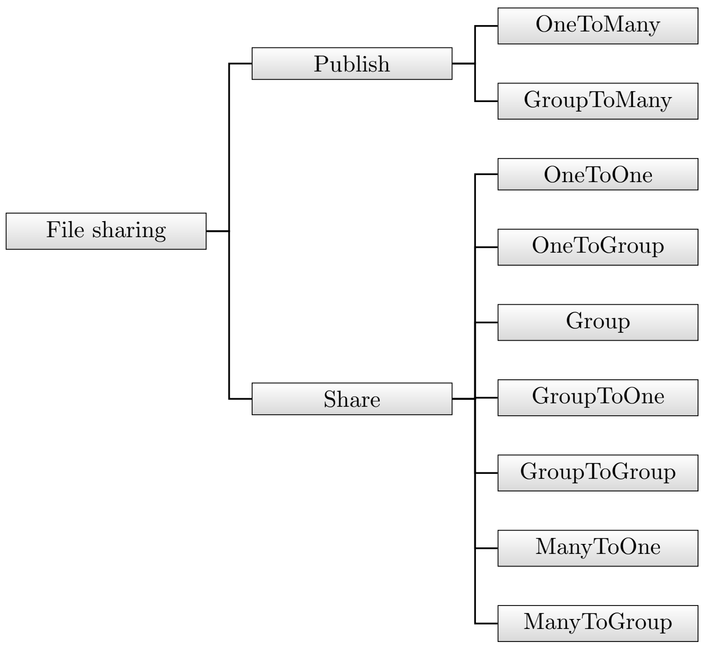

Figure 4.1: How files can be published and shared

to the copies of the original file by the recipients or to the original file by the owner do not reflect on one another. It is useful when the owner of the file does not want to receive a new version of the published or shared files from the recipients or update the copies that the recipients have. An example of a method that allows sharing in a static mode is an email attachment, where neither the owner nor the recipients can observe changes made on the copies of the shared files by others.

Publishing or sharing in Dynamic Mode: This describes a scenario where copies of the original file (that are linked to the original file) are moved from the owner to the recipients. Therefore, any changes made to the copies of the original file by the recipients or by the owner do reflect on one another. This is useful for a collaborative project where a group of members may work on a set of documents collectively. An example of a method that allows sharing in a dynamic mode is Dropbox where a file can be shared and updated by the owner or the recipients, such that both can observe changes made to the copies of the shared files.

Publishing or sharing in Transfer Mode: This describes a scenario where the original file is moved, leaving no copies behind, from the owner to the recipients. The file is treated as a real world object that cannot exist in two places at the same time. Hence, in this mode, releasing a file to more than one recipient requires the file to be held by one recipient at a time. We are not aware of any method that meets this mode of publishing or sharing.

4.2.3 Distributed Memory vs. Shared Memory

Files can be moved from the owner to the recipients directly to their devices or indirectly to a location where recipients can access them (e.g. server). We refer to the former as sharing or publishing in distributed memory (DM), and the latter as sharing or publishing in shared memory (SM). Each of these is described below:

Publishing or sharing in Distributed Memory: A file that is shared or published in DM, will be stored in each recipient's device, allowing them to access the file when they are off-line. DM can be suitable for all sharing or publishing modes (i.e. static, dynamic, and transfer). In the static mode, independent copies of the original files are moved to the recipients' devices, while in the transfer mode, the original file is moved to one recipient's device at a time. In the case of a dynamic mode, copies of the original files are also moved to the recipients' devices; however, the moved copies are linked to the original file, so that any changes made on them will be communicated to other copies.

Publishing or sharing in Shared Memory: A file that is shared or published in SM, will be stored in a central location which recipients must access each time they need to access the file. Thus, unlike DM, a file in a SM requires the recipients to be online to get access to the file. Similar to DM, SM can be suitable for all sharing or publishing modes (i.e. static, dynamic, and transfer). Since the shared or published file is stored in a location which the recipients can access, SM is best for situations where all recipients need to access the same file rather than copies of it. Therefore, in the static mode only a single independent copy of the original file is stored in a location that all recipients can access. In the dynamic mode, a single copy that is linked to the original file is stored in a location that all recipients can access. In the transfer mode, the original file is stored in a location that all recipients can access. Since recipients have access to the same file, changes made to that file will be observed by all recipients without the need to move copies of the file with the new changes to them.

Table 4.1 illustrates 27 types of file propagation. Each cell in the table marked with letter T indicates a way of propagating a file. For example, OneToOne sharing can be performed in static (DM or SM), dynamic (DM or SM) or transfer (DM or SM) mode. In other words, an independent copy of the original file can be moved to one particular recipient's device (static-DM), or to a location that one particular recipient can access

Types of propagation	Static (DM or SM)	Dynamic (DM or SM)	Transfer (DM or SM)
OneToOne	T	T	T
OneToGroup	T	T	T
OneToMany	T	T	T
Group	T	T	T
GroupToOne	T	T	T
GroupToGroup	T	T	T
GroupToMany	T	T	T
ManyToOne	T	T	T
ManyToGroup	T	T	T

Table 4.1: Types of file propagation

(static-SM). A linked copy to the original file can be moved to one particular recipient's device (dynamic-DM), or to a location that one particular recipient can access (dynamic-SM). The original file is moved to one particular recipient's device rather than a copy (transfer-DM), or moved to a location that one particular recipient can access (transfer-SM).

4.3 How files are accessed

Once files are propagated, recipients need to access them. An owner might need to grant the recipients various types of access based on the sharing or publishing purposes. Furthermore, the owner might need to place restrictions on the granted access for further control. In this section we describe the different types of access that might be needed and how each can be restricted.

4.3.1 Types of access

The type of access given to recipients determines the permissibility of two critical operations, which are *read* and *write*. An owner might need to disallow the recipients' read operation to protect the confidentiality of a file, or disallow the recipients' write operation to protect the integrity of a file. The owner might also need to allow both operations if the confidentiality and integrity of a file need not to be protected from the recipients, or to disallow both of them. Furthermore, the write operation is performed to edit a file by either appending a new content to it, or removing content from it. Therefore, an owner might need to allow editing of the file by appending but not removing content from it, or allow both editing of the file by appending and removing content from it. Below we describe six types of access that might be useful in different sharing or publishing scenarios to protect either the confidentiality and/or the integrity of a file.

RO: This type of access allows the recipients of a file to view its content but not to edit it. Therefore, only the read operation can be performed, but not the write operation, and hence, only the integrity of the file is protected from the recipients.

WO$^-$: This type of access allows the recipients of a file to edit its content by appending and removing content from it, but not to view it. Therefore, only the write operation can be performed, but not the read operation, and hence, only the confidentiality of the file is protected from the recipients.

WO$^+$: This type of access allows the recipients of a file to edit the content of the file by appending content to it but not to remove content from it or view it. Therefore, only the write operation can be performed, but not the read operation, and hence, only the confidentiality of the file is protected from the recipients.

RW$^-$: This type of access allows the recipients of a file to view and edit the content of the file by appending and removing content from it. Therefore, both the read and write operations can be performed, and hence, neither the confidentiality nor the integrity of the file are protected from the recipients.

RW$^+$: This type of access allows the recipients of a file to view and edit the content of the file by appending but not removing content from it. Therefore, both the read and write operations can be performed, and hence, neither the confidentiality nor the integrity of the file are protected from the recipients.

NRW: This type of access does not allow the recipients of a file to view or edit the file, but only to hold it. This type of access is useful, for example, when sharing a file with cloud-storage providers. It should be noted that there is a difference between having no type of access at all and having an NRW type of access. The former disallows holding the file, while the latter allows holding the file, but not viewing or editing it.

4.3.2 Restriction on access types

The different types of access described in the previous section might need to be further controlled by an owner, such that an access type granted to the recipients can only be exercised if certain conditions are satisfied. These conditions can be seen as restrictions on the type of access granted to the recipients. We describe four restrictions that can be placed on the granted access type for finer control as follows:

Limited number of times (Ln): This restriction allows the type of access to be exercised for a limited number times. For example, an owner might grant the recipients RO type of access and restrict it to be exercised only for 3 times, after which the file cannot be viewed anymore.

Limited period of time (Lp): This restriction allows the type of access to be exercised for a limited period of time. For example, an owner might grant the recipients RO type of access and restrict it to be exercised only for three days, after which the file cannot be viewed anymore.

Specific time (St): This restriction allows the type of access to be exercised only at a specific time. For example, an owner might grant the recipients RO type of access and restrict it to be exercised only on Monday from 9am to 3pm, but not on any other day or time.

Specific location (Sl): This restriction allows the type of access to be exercised only at a specific location. For example, an owner might grant the recipients RO type of access and restrict it to be exercised only by staff when they are in their offices, but not anywhere else.

Although there might be other types of restriction that can be used to restrict the different types of access, we focused on the minimum set of restrictions that meet the purpose of using them. The purpose of restricting the types of access is not to protect the files against the recipients, since such protection can be realised by the different types of access and propagation mentioned earlier. However, the purpose of such restriction is to protect the files against unauthorised users who acquired a device of a recipient which contains the files. Therefore, such unauthorised users will acquire the type of access given to the recipient, and thus the four types of restriction mentioned above can be used to restrict the type of access given to the recipient to protect the files in such situation. We discuss these restriction in detail in Section 4.5.

Types of access	Ln	Lp	St	Sl
RO	T	T	T	T
WO$^-$	T	T	T	T
WO$^+$	T	T	T	T
RW$^-$	T	T	T	T
RW$^+$	T	T	T	T
NRW	F	T	F	T

Table 4.2: Types of access and restriction

Table 4.2 illustrates 22 types of restricted access that might be granted to the recipients. Each cell marked with letter T indicates a useful type of restricted access that owners of files might need to grant to the recipients. In the table, two cells are marked with letter F to indicate inadequate restriction on an access type. The two inadequate restrictions on the access type are (NRW,Ln) and (NRW,St). This is because the NRW type of access does not allow the recipients to view or edit the file; therefore, limiting the number of times or the specific time to use this type of access is not sensible. However, the other two restrictions on the NRW type of access (i.e. Lp and Sl) might be useful for some sharing scenarios. For example, an owner might need to share files with cloud storage providers, provided that the files are kept in the provider servers that are located at a particular geographical area. Another owner might need the files to be kept in the provider servers until a particular point of time, after which the provider will not be authorised to keep the files in the servers, which must thus be removed.

It should be noted that the recipients can only be granted one type of access; however, various restrictions can be used with that type of access. For example, an owner might grant the recipients the following type of access: (RO, Lp, Sl) which allows the recipients to perform a read operation on the file for a limited period of time and at a specific location. These two restrictions should be satisfied in order for the recipients to view the file.

Table 4.3 combines the different types of files propagation and access; and identifies the useful combinations of these types. The term share and publish can be replaced with any of the categories of file propagation depicted in Figure 4.1.

How files are propagated			How files are accessed																							
			RO				WO-				WO+				RW-				RW+				NRW			
			Ln	Lp	St	Sl	Ln	Lp	St	Sl	Ln	Lp	St	Sl	Ln	Lp	St	Sl	Ln	Lp	St	Sl	Ln	Lp	St	Sl
Share	Static	SM	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	T	F	T
		DM	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	T	F	T
	Dynamic	SM	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	T	F	T
		DM	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	T	F	T
	Transfer	SM	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	T	F	T
		DM	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	T	F	T
Publish	Static	SM	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	T	F	T
		DM	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	T	F	T
	Dynamic	SM	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	T	F	T
		DM	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	T	F	T
	Transfer	SM	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	T	F	T
		DM	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	T	F	T

Table 4.3: Types of files propagation and access

4.4 Taxonomy based on the characterisation of file sharing

Based on the characterisation of the activity of file sharing discussed in the previous section, we define a framework that can be used to classify this activity in a systematic way. This framework, shown in Figure 4.2, will help classify the activity of file sharing by distinguishing how files are propagated to and accessed by the recipients. Below is a brief description of the proposed framework.

The framework has a tree-based structure, where each level represents either a way of propagating files or accessing them. The paths of the tree are numbered; therefore, specifying the path number for each level of the tree starting from the root downwards, will result in a unique class of the activity of file sharing. The first four levels after the root (i.e. paths from 1-18) represent types of file propagation, while the last two levels (i.e. paths 19-28) represent types of file access. At each level of the framework, a unique choice has to be made. In this way, every class of file sharing will form a single path in the tree. However, there is one exception, namely level six -"restriction over access types". Any class of file sharing can utilise multiple restrictions or none (e.g. Ln and Lp at the same time) over one type of access (e.g. RO), as described in the previous section.

To avoid redundant branches, the entire tree is not drawn. For instance, level two has eleven types to choose from; two types belonging to path one and nine types belonging to path two. Each of these types has the same three possibilities for level three (i.e. Static, Dynamic and Transfer). Hence, at level three there are eleven identical groups of the three possible values. Therefore, to avoid using redundant branches, the types at level three are written once and can be used by all types at level two.

The framework, depicted in Figure 4.2, can be utilised in two ways: Firstly, the framework can be applied to classifying the activity of file sharing, by showing different ways that owners might want their files to be propagated and accessed for different sharing scenarios. Secondly, it can be applied to classifying available file sharing methods, by showing which method provides which class of sharing activity. In the next sections, we illustrate how the framework can be applied to classifying both the activity of file sharing in an organisation and some of the popular file sharing methods.

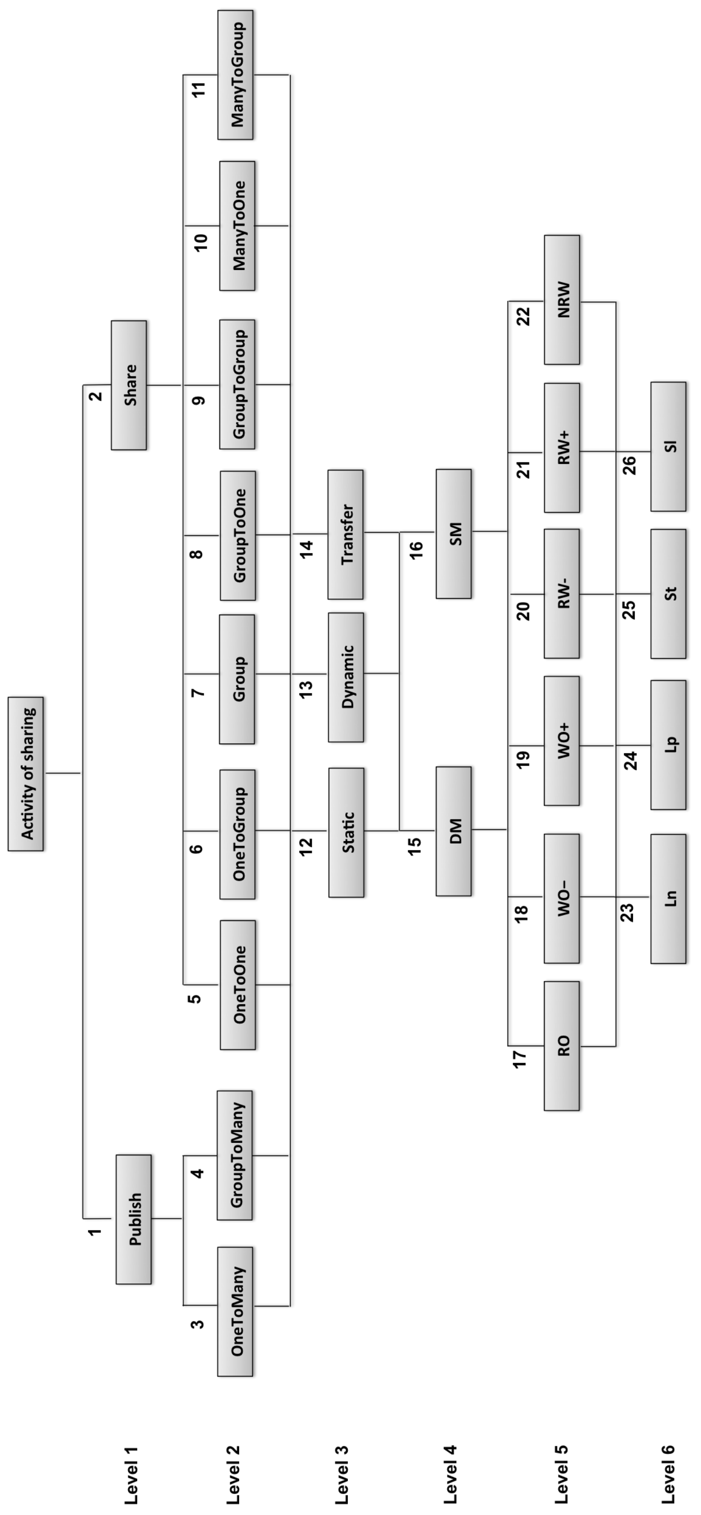

Figure 4.2: Framework for classifying the activity of file sharing

4.4.1 Classifying the activities of file sharing in an organisation.

Alice runs a company that consists of several departments which are Human Resources, Marketing, Production and Finance. Each department contains several employees. Employees within the same department and between different departments need to share files with each other to get their job done. Therefore, Alice wants to define how the activity of file sharing should be performed among employees.

Alice knows the Marketing department is responsible for dealing with customers. The Marketing department sends surveys to customers; however, Alice wants these surveys to be approved by the manager of the department, who is then responsible for moving copies of the surveys to customers' devices, so that customers can read and edit them and return these copies to the department, if they are willing to do so. Hence, Alice has specified the following class of file sharing for this department: 1-3-12-15-20.

Also, Alice knows that employees of the Production department each have to write a report and share it with other employees in the same department; so that each will be aware of others' work and able to modify other reports in the case of mistakes being found. Alice wants employees to view and edit others' reports when they are in their offices and during working hours. Hence, Alice has specified the following class of file sharing for this department: 2-7-13-16-20-(25 + 26).

With respect to the Finance department, Alice knows that employees of this department write reports that are viewed by employees of the Human Resource department, in order for them to make decisions about recruiting new employees. However, Alice wants these reports to be approved by the manager of the department, who is then responsible for moving copies of the reports to the company's server that employees of the Human Resources department can access. This will allow these reports to be updated by the manager of the Finance department, while employees of the Human Resources department will be able to view up-to-date reports. In addition, Alice wants employees of the Human Resources department to view these reports for a limited period of time during working hours. Hence, Alice has specified the following class of file sharing for this department: 2-6-13-16-17(24 + 25).

Finally Alice, who owns the company, needs to view monthly reports written by each department manager. Alice does not want any manager to view reports written by other managers. Therefore, Alice has specified the following class of file sharing among the managers and herself as follows: 2-8-12-15-17.

4.4.2 Classifying file sharing methods

There are various methods of file sharing that exist today. Some of them have been designed merely for sharing files such as File Hosting Services, FTP, and peer-to-peer file sharing; while others are an added feature to the main purpose of applications such as Emails and Social Networking Sites. In this section, some of the most popular file sharing methods are classified in accordance with the taxonomy described in the previous section. The classification is summarised in Table 4.4. Each cell in the table shows which path the sharing method can take at each level of the framework. It should be noted that these methods are classified according to their current features. However, existing features in a file sharing method might be withdrawn and a new feature might be included at anytime, in which case the table should be updated accordingly. Below we give a brief description for each classified method, and show which types of access and propagation the method offers at the different levels of the framework depicted in Figure 4.2.

File sharing methods	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6
Email	2	5, 6, 7, 8, 9, 10, 11	12	15	20	-
Peer-to-peer file sharing	1	3	12	15	20	-
Anonymous FTP	1	3,4	12	15	20	-
None-anonymous FTP	2	5,6,7,8,9	12	15	20	-
Cloud-storage services	1, 2	3, 4, 5, 6, 7, 8, 9	12,13	15	17, 20	-
DFS	2	5, 6, 7, 8, 9	12, 13	16	17, 20	-
File hosting services	1, 2	3, 4, 5, 6, 6, 7, 8, 9, 10, 11	12	15	20	-
Usenet	1	3	12	15	20	-
Instant messaging	2	5, 6, 7, 8, 9	12	15	20	-
Wikis	1, 2	3, 4, 5, 6, 7, 8, 9	12	15	17, 20	-
Blogs	1, 2	3, 4, 5, 6, 7, 8, 9	12	16	17, 21	-
Social networking sites	1, 2	3, 4, 5, 6, 7, 8, 9	12	16	17, 21	-

Table 4.4: Classification of file sharing methods

Email: This is considered the most commonly-used method for sharing files. Although there are a few drawbacks to sharing files via email, such as limitation on file size, it is still a popular method for sharing files at present due to certain features. These features are ease of use, widespread availability and suitability for various tasks. Almost anyone who uses a computer owns an email account, and knows how to use it. Therefore, by using an email to share files, the user will avoid all the difficulties associated with other methods of file sharing, such as ensuring that all recipients have the same method to be able to share the files or ensuring that all recipients know how to use the method of sharing, especially if the method is quite complex and difficult to learn. Examples of emails are Hotmail, Yahoo, and Gmail.

- Level 1: Since email requires the owner of a file to enter the emails addresses of the recipients, which means that recipients should be known in advance, it is only suitable for sharing rather than publishing (i.e. path 2 in Figure 4.2).
- Level 2: Email allows the owner of a file to share it with a particular person or group of people, hence, files can be shared as OneToOne or OneToGroup. A group of owners can share their files with each other by email, as well as sharing them with another person or group. Therefore, files can also be shared as Group, GroupToOne and GroupToGroup. Also, email allows a group of owners who might not be known in advance to the recipients and do not share their files with each other to share them with a particular person or group of people. Therefore, files can be shared as ManyToOne and ManyToGroup. Hence, all paths (i.e. 5,6,7,8,9,10,11 in Figure 4.2) are applicable for sharing files by emails.
- Level 3: Email allows the owner of a file to only send a copy of it to the recipient rather than the file itself. The copy received by the recipient is not linked to the original; therefore, any changes to the copy by the recipient will not be reflected on the original file. Hence, email allows sharing files in the static mode only (i.e. path 12 in Figure 4.2).
- Level 4: Since the copy that is sent to the recipient must be stored in the recipient's device in order to be accessed, email allows sharing files in distributed memory rather than shared memory (i.e. path 15 in Figure 4.2).
- Level 5: Email provides only one type of access to recipients, which is RW^- that allows the recipients to view and edit the received files by appending or removing content (i.e. path 20 in Figure 4.2).

- Level 6: Email provides no restrictions over the type of access that recipients have.

Peer-to-peer file sharing: Peer-to-peer (P2P) file sharing applications have gained much attention in recent years. As its name suggests, P2P file sharing applications utilise a P2P network. Unlike a client-server network, a P2P network consists of multiple computers (nodes) that are able to act as client and server at the same time. For instance, a node in a P2P network can send a request to another node in the network, while responding to requests from other nodes. Therefore, in P2P file sharing applications, files are not uploaded to a central server; instead, they are scattered across users' devices, each of which can act as client and server simultaneously. Examples of P2P file sharing applications are Napster, LimeWire, Shareaza, Kazaa,and BitTorrent.

- Level 1: P2P file sharing requires an owner of a file to use a P2P client to register the file to P2P network. Once the file is registered to the network, other users who use clients that connect them to the same network will be able to search and download that file. Therefore, it is suitable for publishing rather than sharing.

- Level 2: P2P file sharing allows the owner of a file to share it with everyone on the network; therefore, files can be published only as OneToMany.

- Level 3: P2P file sharing allows the owner of a file to publish an independent copy of the file to the recipients. Hence, it allows publishing in the static mode.

- Level 4: Since the sent copies to the recipients will be stored in their devices in order to be accessed, P2P file sharing allows the publishing of files in distributed memory rather than shared memory.

- Level 5: P2P file sharing provides only one type of access for recipients which is RW^- that allows the recipients to view and edit received files by appending or removing content.

- Level 6: P2P file sharing provides no restrictions over the type of access the recipients have.

FTP: This is an acronym that stands for File Transfer Protocol. It is the standard Internet protocol for transferring files from one computer to another. It is not really transferring as in moving files from one location to another, but rather is the copying of files from one computer to another. FTP is an old, but still popular, method for sharing files on the Internet. To share files through FTP, there must be an FTP server which holds all the files to be shared and an FTP client who logs in to the FTP server to obtain file copies. File transfer can occur in two directions as follows: Downloading is transferring a file from an FTP server (the remote computer) to an FTP client (the local computer). Uploading is transferring a file from the FTP client to the FTP server. There are two types of FTP which are anonymous FTP and non-anonymous FTP. Each of these is examined separately below:

Anonymous FTP: This allows anonymous access to the uploaded files on the FTP server to anyone with an FTP client, even through a web browser. Most anonymous FTP servers allow anonymous users to download files from the server, but no one can update the directory except its owner.

- Level 1: Since the files uploaded to the server are publicly available, to be accessed by anyone with an FTP client, anonymous FTP is suitable for publishing rather than sharing.
- Level 2: Anonymous FTP allows the particular owner of files to publish them for everyone; or for a group of owners of files, who share their files with each other, to publish them for everyone. Therefore, files can be published only as OneToMany or GroupToMany.
- Level 3: Anonymous FTP allows the owner of a file to only publish an independent copy of the file to the recipients. Hence, it allows publishing in the static mode.
- Level 4: Since the copies sent to recipients must be stored in their devices in order to be accessed, Anonymous FTP allows files to be published in distributed memory rather than shared memory.
- Level 5: Anonymous FTP provides only one type of access to the recipients which is RW^- which allows the recipients to view and edit the received files by appending or removing content.
- Level 6: Anonymous FTP provides no restrictions over the type of access the recipients have.

Non-anonymous FTP: Unlike anonymous FTP, non-anonymous FTP does not allow anonymous access to the uploaded files on the server. Users accessing a non-anonymous

FTP server will be prompted for a unique username and password which will be used as a basis for making a decision about whether to allow or deny the user access to the files. The differences between anonymous FTP and non-anonymous FTP are only at levels 1 and 2.

- Level 1: Since the users in non-anonymous FTP (unlike anonymous FTP) are prompted for a unique username and password, not everyone can access the files; therefore, non-anonymous FTP is suitable for sharing rather than publishing.
- Level 2: Non-anonymous FTP allows the owner of a file to share it with a particular person or group. Also, it allows a group of owners of files to share them with each other as well as with a particular person or group; or a group of owners who might not be known in advance to the recipients and do not share their files with each other to share them with a particular person or group. Therefore, files in non-anonymous FTP can be shared as OneToOne, OneToGroup, Group, GroupToOne, GroupToGroup, ManyToOne, and ManyToGroup.

Cloud-storage services: These allow users to create storage accounts to store their files. Users are able to perform several operations on their storage accounts such as upload, download, delete, and to share files. These operations can be performed by the users in two ways. Firstly, through a web browser from any device; and secondly, through a proprietary software client installed into their devices. Cloud-storage services offer a synchronisation service, which means operations on a storage account made through a browser will be redirected in the installed client of that account and vice versa. Also, users who own several devices (e.g., laptop, tablet, Smartphone) can install a client into each device to synchronise the files stored in their storage accounts across their devices. Examples of cloud-storage services are Dropbox, Google Drive and Microsoft's Skydrive.

- Level 1: Cloud-storage services allow users to share their files either with users subscribed to the same service or with users from the outside. Sharing files with other users subscribed to the same service requires the owner of a file to select a person or group of people from the same service to share the file with and specify the operations that they can perform on the shared file (e.g., read and write). Since the file will be released only to users from the same service who the owner has selected, it is suitable for sharing. On the other hand, sharing files with other users that are not subscribed to the same service requires the owner of the files to generate a URL for that file and distribute the URL to others. The URL can be distributed to everyone (e.g. posted in a public forum) or to a person or group (e.g. via email). Therefore, Cloud-storage services are suitable for publishing and sharing.

- Level 2: Cloud storage services allow the owner of a file or group of owners of files who are sharing them with each other to publish their files for everyone. Also, it allows the owner of a file to share it with a particular person or group and a group of owners to share their files with each other as well as sharing them with a particular person or group. Therefore, files in cloud storage services can be published or shared as OneToMany, GroupToMany, OneToOne, OneToGroup, Group, GroupToOne, GroupToGroup.

- Level 3: Cloud storage services allow the owner of a file to publish or share an independent or linked copy of the file with recipients. Hence, it allows publishing and sharing in the static and dynamic modes.

- Level 4: The published or shared copies of the files must be stored in the recipients' devices to be accessed. Therefore, Cloud-storage services allow the publishing and sharing of files in distributed memory.

- Level 5: Cloud-storage services allow the recipients to have only RO and RW$^-$ types of access.

- Level 6: Cloud-storage services provide no restrictions over the type of access that recipients have.

Distributed file systems: These are file systems that allow and manage access to files and folders from multiple computers through a network. They are similar to traditional file systems but designed to provide file storage and controlled access to files over local and wide area networks. In DFS, files are stored on one or more central servers that can be accessed by any number of remote clients in the network. The remote clients can retrieve the files from the server to work with them as if they were stored locally on their computers.

To protect files, DFS utilises authentication and authorisation techniques. The former is used to allow only authorised users to access the files; while the latter is used to specify the operations that authorised users are allowed to perform on the accessed files, such as reading, writing, and deleting. Authentication is often implemented as a username/password, symmetric key cryptography (e.g., Kerberos) or public key cryptography (e.g., X.509) whereas authorisation is often implemented as an Access Control List (ACL).

Unlike other file sharing methods which are mainly focused on transferring the files from one location to another, DFS provides other features that enhance the activity of sharing; namely, users' mobility, files' availability and transparency. Users' mobility means that a user can store his/her files in a server and access these files in a uniform view from any computer. This is very useful in environments where users do not have a particular computer to work on, so that they can use any computer to access their files. Files' availability means that the files will be available to access all the time, even if the computer of the user has crashed due to software or hardware failure. This is because the files are not stored locally, so the user can use another computer to access the files, which are stored on the server. Even if a server in DFS has crashed, the files will be available as DFS utilises replication techniques to spread the files to multiple servers and thus avoid single points of failure. Transparency means that users will be able to access files over a network as easily as if the files were stored locally. DFS is designed to be transparent in different aspects including login, access, location, concurrency, failure, and replication, which results in remote clients not being aware of any systemic complexity and only seeing a system that is similar to a local file system.

DFS can be implemented as a client-server network or peer-to-peer (p2p) network. Examples of the former are NFS [99], AFS [49], SMB [70], Coda [102], and xFS [7]; while examples of the latter are Ivy [72], Farsite [2], and OceanStore [59]. Although p2p DFS utilises the same underlying techniques as a p2p file sharing application, the main difference between them is that p2p DFS provides persistent non-volatile storage with a file system like interface. This interface provides a layer of transparency for the user and to the applications which access it.

- Level 1: Since DFS requires clients who are recipients to authenticate themselves with the system before accessing the files (i.e. only authorised users can access the files), it is suitable for sharing rather than publishing.

- Level 2: DSF allows a particular person to share his files with another particular person or group of people; and allows a group of people to share their files with each other as well as sharing their files with a particular person or a group of people. Therefore, files can be shared as OneToOne, OneToGroup, Group, GroupToOne and GroupToGroup.

- Level 3: DFS allows the owner of a file to share an independent or linked copies of the file with the recipients. Hence, it allows sharing in both static and dynamic modes.
- Level 4: The shared copies of the files must be stored in a central server that need to be accessible to the recipients. Therefore, DFS allows sharing files in shared memory.
- Level 5: DFS allows the recipients to have two types of access which are RO and RW^- .
- Level 6: DFS provides no restrictions over the type of access that the recipients have.

File hosting services: This type of file sharing is also known as a one-click hosting service. It has recently gained much popularity, as it provides easy steps to share files compared to other types of file sharing methods. These easy steps are as follows: Firstly, the user uploads any type of file to the server of the file hosting service through a basic web interface. Secondly, the file hosting service provides the uploader with a URL for the file. Thirdly, the uploader shares the URL with other people either privately, via email for example, or publicly through posting the URL on any public sites. Originally this type of file sharing was designed for file backup purposes and for uploading a file that was too big to be sent as an email attachment. Examples of file hosting services are Rapidshare, Hotfile, zSHARE, and Mediafire.

- Levels 1 and 2: Since file hosting services only generate a URL for the files to be downloaded by the recipients, classifying it depends on how the URL is shared. The URL can be shared using one or more of the other file sharing methods. However, whatever method of sharing is used to share the URL, file hosting services will inherit the characteristics of that method. For example, if the URL is shared by using email, the file hosting service will inherit the characteristics of levels 1 and 2 of email, and if the URL is shared using anonymous FTP, then it will inherit the characteristics of levels 1 and 2 of anonymous FTP.
- Level 3: File hosting services allow the owner of a file to publish or share an independent copy of the file with the recipients. Hence, it allows publishing or sharing in the static mode.
- Level 4: The published or shared copies of the files must be stored in the recipients'

devices in order to be accessed. Therefore, file hosting services allow the publishing and sharing of files in distributed memory.

- Level 5: File hosting services allow the recipients to only have RW^- type of access.
- Level 6: File hosting services provide no restrictions over the type of access the recipients have.

Usenet: This is a collection of newsgroups where users can post messages and files that are distributed among multiple servers known as news servers, NNTP servers or news-feeds. Unlike p2p file sharing applications which utilise p2p networks, Usenet utilises the traditional client-server network. Therefore, instead of users searching and downloading files directly from each other's devices, as is the case in p2p file sharing applications, they search and download files from a News server. In other words, the files must first be uploaded to a News server, which will distribute them to other News servers that users can connect to in order to search for and download the uploaded files.

- Level 1: In order for a file to be shared, the users must use a client called news-reader that allows them to connect to a News server to upload their files and search for and download files uploaded by other users. Once the owner of the files has uploaded his files to a newsgroup in a News server using a newsreader, other users can use their newsreaders to connect to any News server to search for and download these files. Therefore, Usenet is suitable for publishing rather than sharing, as the files will be shared with everyone using Usenet.
- Level 2: Usenet allows a particular person to share his files with everyone. Therefore, Usenet allows files to be published only as OneToMany.
- Level 3: Usenet allows the owner of a file to publish an independent copy of the file for the recipients. Hence, it allows publishing in the static mode.
- Level 4: The published copies of the files must be stored in the recipients' devices to be accessed. Therefore, Usenet allows files to be published in distributed memory.
- Level 5: Usenet allows the recipients to only have RW^- type of access.
- Level 6: Usenet provides no restrictions over the type of access the recipients have.

Instant messaging: This is a form of online communication that allows real-time interaction through personal computers or mobile computing devices. It allows people to exchange messages with each other. In addition to exchanging messages, which is its main function, IM allows people to exchange files. IM can be implemented as a Client-server network or Peer-to-peer network. In the former, the client communicates with the IM sever to locate other users and exchange messages. Messages are not sent directly from the sender to the receiver, but are sent first to the IM server and then from the IM server to the receiver. In the latter, the client only contacts the IM server to locate other clients. Once the client has located its peer, it contacts the peer directly. For transferring files, most systems transfer them directly among clients rather than through the IM server. Examples of IM are ICQ, AOL Instant Messenger, Skype, Paltalk, Google Talk and Yahoo Messenger.

People often do not differentiate between the terms *chat* and *IM*. Although both of them allow users to send short messages to each other in real time, the two terms convey different meanings. In IM, in order for a user to communicate with others, the user must first add them to his list of contacts, called the Buddy List or Friend List. This list allows the user to choose who he wants to communicate with. The user will be able to communicate with only one person on his list of contacts at a time. Alternatively, he can create a private chat room (also known as a group chat) and invite more than one user from his list of contacts, so that other users can join the private room by invitation from any of the existing members of the private chat room. On the other hand, chat does not impose such lists of contacts and often occurs in a virtual public chat room consisting of many different users who may or may not know each other, for the purpose of discussing a particular topic of interest. Most IM service providers, such as Paltalk, ICQ, Skype and AOL Instant Messenger, integrate public chat rooms and other features such as Voice and Video chat to their IM services. However, file sharing can only occur in IM but not in public chat rooms.

- Level 1: Since IM requires the users to add others to their list of contacts (or their friends' lists of contacts in the case of sharing files in a private chat room) before sharing takes place, IM is suitable for sharing rather than publishing.

- Level 2: IM allows an owner of a file to share his file with a particular person or group, and a group of owners to share their files with each other as well as sharing their files with a particular person or a group of people. Therefore, files in IM can be shared as OneToOne, OneToGroup, Group, GroupToOne, and GroupToGroup.

- Level 3: IM allows the owner of a file to share an independent copy of the file with the recipients. Hence, it allows sharing in the static mode.
- Level 4: The shared copies of the files must be stored in the recipients' devices to be accessed. Therefore, IM allows the sharing of files in distributed memory.
- Level 5: IM allows the recipients to only have RW^- type of access.
- Level 6: IM provides no restrictions over the type of access the recipients have.

Wikis: A wiki is a webpage or set of webpages that can be viewed and edited by anyone who is allowed access. In wikis, users can create a webpage and add content to it; other users can view this content and edit the page by modifying already existing content or adding new content. While text is the primary content in wiki pages, users are able to add photos, audios or videos to the pages, or put in links to other files that cannot be displayed on the pages. Wikis can be public which means that webpages are available to anyone on the Internet, or private, which means that webpages are only available to selected individuals. A well-known example of a public wiki is Wikipedia and of a private wiki is SamePage.

- Level 1: Since public wikis allow anyone to view their webpages, and private wikis allow only selected individuals to view their webpages, wikis are suitable for publishing and sharing.
- Level 2: Although public and private wikis allow their webpages be viewed and edited by anyone who is allowed access, they can implement access control based on username and password to restrict who can view and edit which pages. However, such access control can only be implemented by the owner of the wiki itself but not the users who access the wiki. Based on how access control is used in public and private wikis, wikis can allow a particular person to share his files with another particular person, a group of people, or everyone. Also, it allows a group of people to share their files with each other, as well as with a particular person, a group of people, or everyone. Therefore, wikis allow publishing and sharing files as OneToOne, OneToGroup, OneToMany, Group, GroupToOne, GroupToGroup, and GroupToMany.
- Level 3: Wikis allow the owner of a file to publish or share an independent copy of the file with recipients. Hence, it allows publishing and sharing in the static mode.

- Level 4: The published and shared copies of the files must be stored in a webpage to be accessed by recipients. Therefore, wikis allow the publishing and sharing of files in shared memory.
- Level 5: Wikis allow the recipients to have two types of access, namely, RO and RW^- .
- Level 6: Wikis provide no restrictions over the type of access the recipients have.

Blogs: *Blog* is an abbreviated term for weblog which is a webpage or set of webpages that are created and owned by a user for others to view and edit.

- Level 1: Similar to wikis, blogs can be public or private, and users can add text, photos, videos or audios to their own webpages, or links to other files that cannot be displayed on the pages. Public blogs allow the webpages created by the users to be available to everyone on the Internet; and private blogs allow the webpages to only be available for selected individuals. Hence, blogs are suitable for publishing and sharing.
- Level 2: A blog can be owned by a single person or a group of people. If a blog is owned by a single person, webpages can only be created by that person; whereas if the blog is owned by a group of people, webpages can only be created by someone from that group. Therefore, if a blog is public and owned by a single person, files can be published as OneToMany, while if the blog is public and owned by a group of people, files can be published as GroupToMany, On the other hand, if the blog is private and owned by a single person, files can be shared as OneToOne and OneToGroup, while if the blog is private, and owned by a group of people, files can be shared as Group, GroupToOne and GroupToGroup.
- Level 3: Blogs allow the owner of a file to publish or share an independent copy of the file with recipients. Hence, it allows publishing and sharing in the static mode.
- Level 4: The published and shared copies of the files must be stored in a webpage to be accessed by recipients. Therefore, Blogs allow the publishing and sharing of files in shared memory.
- Level 5: Blogs allow the recipients to have two types of access, namely, RO and RW^+ .
- Level 6: Blogs provide no restrictions over the type of access the recipients have.

Social networking sites Like blogs, social networking sites allow users to create and own webpages for others to view and edit.

- Level 1: Unlike blogs, users who access the webpages created by others can edit these pages by adding new content of any type (e.g. a photo, video, audio, text or links). Like wikis and blogs, the webpages in social networking sites can be public or private, and users can add texts, photos, videos or audios to their webpages or links to other files that cannot be displayed on the pages. Therefore, social networking sites are suitable for publishing and sharing.
- Level 2: Like blogs, files can be published and shared as OneToMany, GroupToMany, OneToOne, OneToGroup, Group, GroupToOne and GroupToGroup.
- Level 3: Like wikis and blogs, social networking sites allow publishing and sharing in the static mode.
- Level 4: : Like wikis and blogs, social networking sites allow publishing and sharing files in shared memory.
- Level 5: Social networking sites allow the recipients to have two types of access which are RO and RW$^+$.
- Level 6: Social networking sites provide no restrictions over the type of access the recipients have.

4.5 Discussion

The classification framework of the activity of file sharing described in this chapter can be thought of as a series of policies that describe how files should be propagated and accessed for different sharing scenarios. Enforcing these policies provides the protection required against the accidental misuse described in the previous chapter. In particular, the type of access given to the recipient can be used to prevent accidental disclosure and modification by a naive. The types of restriction on the various access types can be used to prevent accidental disclosure and modification by a masquerader; and the types of propagation can be used to prevent accidental redistribution by a naive and a masquerader. In this section we discuss how these policies can be used to protect files against accidental misuse.

Protecting the confidentiality of files at the recipients from accidental disclosure to a naive: Accidental disclosure of shared files to a naive occurs when legitimate recipients view a file that they are not allowed to view. There are three types of access that can be used to specify policies to control read operations in order to protect the file against accidental disclosure to a naive. These types of access are NRW, WO$^-$ and WO$^+$. The NRW type of access is suitable when the recipient is only allowed to hold the file, since read and write operations are not allowed with this type of access. The WO$^-$ type of access is suitable when the recipient is only allowed to edit the file by appending or removing content, but not to view it, since read operations are not allowed in this type of access. The WO$^+$ type of access is suitable when the recipient is only allowed to edit the file by appending content, but not by removing or viewing it; since in this type of access read operations are not allowed and write operations only allow editing of the file by appending content, but not by removing content from it.

Protecting the integrity of files at the recipients from accidental modification by a naive: Accidental modification of the shared files by a naive occurs when a legitimate recipient edits a file that is not allowed to be edited, or edits a file in an unauthorised manner. There are four types of access that can be used to specify policies that control write operations in order to protect the file against accidental modification by a naive. These types of access are NRW, RO, WO$^+$ and RW$^+$. The NRW type of access is suitable when the recipient is only allowed to hold the file, since read and write operations are not allowed with this type of access. The RO type of access is suitable when the recipient is only allowed to read the file, since write operations are not allowed with this type of access. The WO$^+$ type of access is suitable when the recipient is only allowed to edit the file by appending, but not removing content from it, since write operations are only allowed to append content to the file with this type of access. The RW$^+$ is type of access is suitable when the recipient is only allowed to view and edit the file by appending, but not removing content from it, since write operations are only allowed to append content to the file with this type of access.

Protecting the confidentiality and integrity of files at the recipients from accidental disclosure to and modification by a masquerader: Accidental disclosure to and modification by a masquerader occurs when the device of a legitimate recipient, which contains the shared file, is acquired by an unauthorised user. The masquerader in this case will acquire the same type of access that is given to the recipient. Therefore, if the file was protected against accidental disclosure to and modification by a naive, then it will be protected against the masquerader as well. For example, if an NRW type of access was given to the recipient, the confidentiality and integrity of the file will not be violated by a masquerader. However, it is more challenging to prevent a masquerader from exercising other types of access given to the recipient, such RO and WO^+. The different types of restriction on the type of access given to the recipient can be used to specify policies that control read and write operations to protect the files against accidental disclosure to and modification by a masquerader.

Limiting the number of times (Ln) and the period of time (Lp) to exercise a specific type of access is useful when the owner knows that the recipient does not need to exercise that type of access indefinitely on the shared file. Such restrictions are specified with the hope that they will not be satisfied when a masquerader acquires the device from the legitimate recipient; i.e. that the access has reached its limited number of times or period of time, and thus cannot be exercised. Although there is a chance that a masquerader acquires the device from legitimate recipient, while the restrictions are still satisfied, the consequences of violating the access will be less than having no restriction at all.

Specifying a specific time (St) and location (Sl) to exercise a specific type of access is useful when the owner knows that the recipient needs access indefinitely but not all the time and in every location. Such restrictions are specified with the hope that the device will be secure against access by a masquerader during the specified time and in the specified location.

Specifying these restrictions altogether provides a strong protection against accidental disclosure to and modification by a masquerader. However, there are situations where some of these restrictions cannot be specified. For instance, Alice might not know how many times Bob needs to view her report, However, she might know for how long Bob needs view her report. Also, Alice might not know at which time Bob needs to view her report, However, she might know that Bob needs to view her report from his office etc. It should taken into account that the more of these restrictions are specified, the fewer chances a masquerader has to disclose or modify the shared files accidentally.

Protecting the shared of files at the recipients from accidental redistribution by a naive or masquerader: Accidental redistribution of the shared files occurs when the file is sent to an unauthorised user by a naive or by a masquerader who acquires the device of a legitimate recipient. The different types of file propagation can be used to specify policies that control send operations to protect the file against accidental redistribution. For example, specifying a type of file propagation such as (Alice $\rightarrow$ Bob - Static - DM), will only allow the send operation to be performed if a copy of the file that is not linked to the original file is sent by the owner Alice to the recipient Bob or by the recipient Bob to the owner Alice. Specifying a type of file propagation such as (Alice $\rightarrow$ {Bob,Carol,Dave} - Transfer - SM), will only allow a send operation to be performed if the original file, and not a copy, is sent by the owner Alice to a location which the recipients {Bob,Carol,Dave} can access; or is sent by any of the recipients {Bob,Carol,Dave} to a location which the owner Alice or any of the recipients can access. Since the types of file propagation specify to whom the file can be sent, accidental redistribution by owners, recipients and masqueraders will be prevented. This is because in all cases the file can only be sent to the users specified in the policy. For example, even if a masquerader acquires the device of the owner or recipients, the masquerader will only be able to send the files to authorised recipients specified in the policies but to no one else.

4.6 Summary

In this chapter, we have characterised the activity of file sharing based on two factors: how files can be propagated from owners to recipients; and how files can be accessed by the recipients after their propagation. Based on the characterisation of the activity of file sharing, we defined a framework that classifies the activity of file sharing into different categories. These categories can be thought of as policies that describe how files should be propagated and accessed in ways that satisfy different sharing scenarios. The framework can be applied to the classification of available file sharing methods to find out which method supports which categories of sharing. In such a way, users will choose the appropriate method of sharing that supports the category of sharing needed. Enforcing the different policies identified in the framework plays an important role in mitigating accidental misuse of the shared files. In the next chapter, we present a novel approach for enforcing these policies.

Chapter 5

Secure file system

5.1 Introduction

In this chapter we propose a novel approach to enforce the policies discussed in the previous chapter. We follow a language-based technique to enforce these policies, particularly by the use of a type system. We design a language of commands to manipulate files and specify their policies in a Unix-like file system, and a type system to enforce these policies. In this setting, files are associated with security types that represent security policies, and programs are sets of commands to be issued on files such as *read*, *copy*, *move*, etc. The type system plays the role of a reference monitor that intercepts and statically analyses each command to be issued on a file and determines whether or not the command is safe to be executed. Safe commands are those which do not cause errors during execution. Such errors might be caused by commands that violate the security policies associated with the files or violate its own requirements (e.g. a file must exist to be removed). Therefore, if commands are type-checked, then files' and commands' policies are not violated and can be executed safely.

In this chapter, we focus on enforcing a particular constraint of the policies discussed in Chapter 4; namely, limiting the number of times a file can be read in a shared-memory style. This is because the ideas developed in this chapter to enforce this constraint represent the basic building blocks of our approach which can be easily extended to enforce the whole policies discussed in Chapter 4. We discuss extensions to enforce these policies in Chapter 6.

The contribution of this chapter is not to provide a thoroughly secure file system, but rather is to provide a security mechanism to secure file sharing which is performed through issuing various commands on files such as *read*, *copy*, and *move* by various users in a file system. The objective of the security mechanism is to protect shared files against commands issued to manipulate them. Therefore, the notion of security we are concerned with is that of file sharing, rather than the file system as a whole. In this chapter, we focus on a small set of commands that manipulate files, and we present our approach to secure files against them.

The rest of this chapter is organised as follows: in Section 5.2 we present the notations that we use throughout this chapter. In Section 5.3 we define security types that represent policies to regulate *copy* operations. These security types control the access to *copy* operations and the flow caused by all operations including *copy*, such that policies for copying files are not violated. In Section 5.4 we present a language that enforces these policies, and define its syntax and semantics. We define the language semantics as small-step and big-step semantics, and we show they are equivalent. In Section 5.5 we define the security errors that we aim to prevent in our language. We divide these errors into syntactical and type errors and describe each of them. In Section 5.6 we discuss syntactical errors and define an algorithm to check for the syntactical correctness of commands before execution. In Section 5.7 we present our type system that prevents errors which might occur during execution, whether syntactical or type errors. In Section 5.8 we prove the soundness of our type system with respect to the language semantics. In Section 5.9 we follow the method of Hindley-Milner [27] and define a type inference algorithm and prove its soundness and completeness. Finally, we summarise this chapter in Section 5.10.

5.2 Notations

A file system is represented as a set of files and ranged over by Greek small letters. A file has a name, content and a type, and we write $f(c) : \tau$ for a file with name f, content c and type τ. The type τ associated with a file will serve as a permission to manipulate the file in accordance with the type. In this chapter we deal with three kinds of sets of files.

- The set of files with names, content and types, ranged over by Greek small letters $\delta, \gamma, \ldots$. For example, $\delta = \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2, \ldots, f_n(c_n) : \tau_n\}$. Therefore, it represents the whole file system.

- The set of files with names and types but not content, ranged over by Greek capital letters $\Gamma, \Delta, \ldots$. For example, $\Gamma = \{f_1 : \tau_1, f_2 : \tau_2, \ldots, f_n : \tau_n\}$.

- The set of file names only, ranged over by Roman capital letters $H, N, E, \ldots$. For example, $H = \{f_1, f_2, \ldots, f_n\}$.

Throughout this chapter, we sometimes need to compare two sets of different structures, such as $H = \delta$ where we are only interested in the name part of the files in the sets, or $\Gamma = \delta$ where we are only interested in the name and type parts of the files in the sets. Also, sometimes we are interested in the name part of the files in the set δ and Γ to check whether or not a particular file name exists in them, regardless of its type and content. To facilitate this, we introduce the functions e^c and e^t which are applied to the sets δ and Γ. The function e^c takes the set δ and erases the content of files in the set, whereas the function e^t takes the set δ or Γ and erases the types of files in the set. For example, if $\delta = \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2\}$ and $\Gamma = \{f_1 : \tau_1, f_2 : \tau_2\}$, then $e^c(\delta) = \{f_1 : \tau_1, f_2 : \tau_2\}$, $e^t(\delta) = \{f_1(c_1), f_2(c_2)\}$, and $e^t(\Gamma) = \{f_1, f_2\}$. We apply these functions to sets to extract the parts of files we are concerned with. For example, $f_1 \in e^t(\Gamma) \wedge e^c(e^t(\delta))$, whereas $f_5 \notin e^t(\Gamma) \wedge e^c(e^t(\delta))$, and if $H = \{f_1, f_2\}$, then $H = e^t(\Gamma) = e^c(e^t(\delta))$. For simplicity, in this chapter we do not write these functions explicitly, rather we assume they are applied to sets where appropriate. Therefore, instead of writing $H = e^t(\Gamma) = e^c(e^t(\delta))$ and $f_1 \in e^t(\Gamma) \wedge e^c(e^t(\delta))$, we simply write $H = \Gamma = \delta$ and $f_1 \in \Gamma \wedge \delta$, respectively. It should be clear from the context which parts of the files in a set we are concerned with. We use this convention for all set operations (e.g. $\subseteq, \cup, \cap, \ldots$) rather than just equality.

5.3 Security types and policies

Our approach to limiting the number of times a file can be read is by limiting the number of copies the file can produce. Therefore, in this section we define policies to regulate copy operations on files. To control the access to `copy` operations on files we define three security types which are UC, LC^n, and NC each of which specifies a distinct policy of how `copy` operations can be performed on them. We refer to such types as *security copy types.* UC stands for Unrestricted Copy, which means that a file associated with this type can be copied without restriction. The copied version of a file of type UC should be allowed to be copied in the same way, so should also be of type UC. LC^n stands for Linear Copy, which means that a file associated with this type can be copied n number of times, after which the file cannot be copied anymore. However, unlike UC, the copied version of a file of type LC^n should not be copied anymore. NC stands for No Copy, which means that a file associated with this type cannot be copied at all. Hence, the copied version of a file of type LC^n should be of type NC. To formally state the above policies, we define the following functions and notations on types. The function *dst* stands for destination, for a given type of a file, the function *dst* finds the appropriate type for the copied version of that file.

$$dst(\text{UC}) = \text{UC} \qquad dst(\text{LC}^n) = \text{NC} \iff n > 0.$$

The function *red* stands for reduction, for a given type of a file, the function *red* reduces that type if needed when it is copied. It is mainly useful for the type LC^n to limit the number of times the type can be copied.

$$red(\text{UC}) = \text{UC} \qquad red(\text{LC}^n) = \text{LC}^{n-1} \iff n > 0$$

Let $T(f)$ denote the type associated with the file f. Then the policies of *security copy types* described above can be stated as follows: a file f can be copied if and only if $T(f) \in \{\text{UC}, \text{LC}^{n>0}\}$ and the copied version of f must have the type $dst(T(f))$ and f must have the type $red(T(f))$ after it has been copied. For example, assume that $T(f) = \text{LC}^2$, then f can be copied since $\text{LC}^2 \in \{\text{UC}, \text{LC}^{n>0}\}$ and once is copied, f must have the type $red(\text{LC}^2) = \text{LC}^1$ and the copied version of f must have the type $dst(\text{LC}^2) = \text{NC}$. Note that we do not define $dst(\text{LC}^{n\leq 0})$ nor $dst(\text{NC})$ or $red(\text{LC}^{n\leq 0})$ and $red(\text{NC})$. This is because files of these types do not satisfy the condition $\text{NC}, \text{LC}^{n\leq 0} \in \{\text{UC}, \text{LC}^{n>0}\}$, thus cannot be copied.

However, some operations other than `copy` might cause information to flow from a file to another. Let $f_1 \rightarrow^o f_2$ denotes flow of information from f_1 to f_2 by operations other than `copy` such as `mv`, `cat`, etc. Such a flow of information might violate the copy policies of files. For example, assume $T(f_1) = \text{NC}$ and $T(f_2) = \text{UC}$, then $f_1 \rightarrow^o f_2$, will lead the file f_1 to be copied indirectly without any restriction by copying f_2. To control the information flow among files, our security copy types form a lattice $(\tau, \sqsubseteq)$, where $\tau = \{\text{NC}, \text{LC}^n, \text{UC}\}$, are partially ordered by $\sqsubseteq$ (see Figure 5.1). NC and UC are the upper bound and the lower bound of the set τ, respectively. The least restrictive type is UC, while the most restrictive type is NC. Therefore, information is allowed only to flow upwards in the lattice, which means from the less restrictive type to the more restrictive. It should be noted that a type LC^n is less restrictive than a type $\text{LC}^{n'}$ if and only if $n > n'$. For example, LC^4 is less restrictive than LC^2, therefore, information is allowed to flow from a file of type LC^4 to a file of type LC^2.

By having a lattice of security types, there are two kinds of information flow policies that can be enforced based on whether the type system is flow-insensitive or flow-sensitive. The policy enforced by flow-insensitive type systems is inappropriate when the security types represent access permissions to operations. This might not be true for the current security copy types we address in this chapter. However, when we discuss the additional

$$\begin{tikzcd}
\mathrm{NC} \arrow[d, no head] \\
\mathrm{LC}^n \arrow[d, no head] \\
\mathrm{UC}
\end{tikzcd}$$

Figure 5.1: Security copy types

types to control access to other operations in Chapter 6 such inappropriateness will be obvious. On the other hand, the policy enforced by flow-sensitive type systems is quite promising to control the flow of information among files. However, our view is that each file should have its own security policy which should be respected regardless of the information flowed into it, and only allowed to change to a more restrictive policy.

Therefore, the information flow policy we need to enforce is somewhere in between the flow policies enforced by flow-insensitive and flow-sensitive type systems. We follow the idea of flow-insensitive type systems in that flow of information must only result in a more restrictive type of information, while we follow the idea of flow-sensitive type systems in that information can flow anywhere, and the security types can be changed during computation. This can be achieved by allowing information to flow from a security type τ_1 to any security type τ_2, provided that the security type τ_2 is changed to the least upper bound of τ_1 and τ_2 (i.e. $\tau_1 \sqcup \tau_2$), after the flow of information. Since $\forall \tau, \tau' \in \mathcal{T},\ \tau \sqsubseteq \tau \sqcup \tau'$, where $\mathcal{T}$ is lattice of security types, any information flow is considered a restriction as long as the destination changes its type to the least upper bound of its type and the source type. This is because the least upper bound of two types is always more restrictive than each of them. In such way we benefit from the restrictiveness of flow-insensitive type systems and the permissiveness of flow-sensitive type systems.

However, even if information flows to a destination file of type that is at least as restrictive as the type of the source file, the copy policy of the source file might still be violated. For example, assuming that $T(f_1) = \mathrm{NC}$ and $T(f_2) = \mathrm{UC}$, then $f_1 \to^o f_2$ should result in f_2 changing its type to $T(f_1) \sqcup T(f_2) = \mathrm{NC}$. However, now the information exists in both the source file f_1 and the destination file f_2, and thus, $f_1 \to^o f_2$ has the same effect as copying f_1 to f_2 even though the copy policy of f_1 does not allow it. Another example, assuming $T(f_1) = \mathrm{LC}^4$ and $T(f_2) = \mathrm{UC}$, then $f_1 \to^o f_2$ should result in f_2 changing its type to $T(f_1) \sqcup T(f_2) = \mathrm{LC}^4$. However, now the information exists in both the source file f_1 and the destination file f_2, and thus, f_1 can be copied directly 4 times and also indirectly 4 more times by copying f_2 even though the policy of f_1 does not allow it. We overcome this violation by the notion of resource consumption, that is a file is consumed when it is used. Therefore, information flow such as $f_1 \rightarrow^o f_2$, will consume f_1 and allow only f_2 to exist after the flow. In such a way, any information flow is a restriction and will never violate the copy policies of files.

Therefore, information flow such as $f_1 \rightarrow^o f_2$ is always allowed, provided that f_2 changes its type to $T(f_1) \sqcup T(f_2)$ and f_1 is consumed after performing the operation. However, this is useful when f_2 is associated with a type. Operations such as $f_1 \rightarrow^o f_2$ can be performed while f_2 is not associated with a type. In such case, it is sufficient to assign the type of f_1 to f_2, that is $T(f_2) = T(f_1)$. Let $f \in \mathit{types}$ denotes a file f has a type, and $f \notin \mathit{types}$ denotes a file f does not have a type either because it does not exist, and therefore, it must be created for the flow to proceed, or might not be associated with any policy. Below we define the policies for performing operations other than `copy`.

Definition 5.3.1. $\forall f_1, f_2 \in \mathit{types}.\ f_1 \rightarrow^o f_2$ is allowed, provided that f_2 must change its type to $T(f_1) \sqcup T(f_2)$ and f_1 is consumed after performing the operation.

Definition 5.3.2. $\forall f_1 \in \mathit{types} \land \forall f_2 \notin \mathit{types}.\ f_1 \rightarrow^o f_2$ is allowed, provided that f_2 must be assigned the type $T(f_1)$ and f_1 is consumed after performing the operation.

The above definitions show the constraints on operations that cause flow of information from a single source file to a destination file. However, some operations might cause flow of information from multiple source files to a destination file. Let $f_1, f_2 \rightarrow^o f_3$ denotes an operation that causes flow of information from f_1 and f_2 to f_3. In this case, if $f_3 \in \mathit{types}$, then the type of f_3 must be changed to $T(f_1) \sqcup T(f_2) \sqcup T(f_3)$, whereas if $f_3 \notin \mathit{types}$, then f_3 must be assigned the type $T(f_1) \sqcup T(f_2)$. We give the following definitions for such cases as follows.

Definition 5.3.3. $\forall f_1, f_2, f_3 \in \mathit{types}.\ f_1, f_2 \rightarrow^o f_3$ is allowed, provided that f_3 must change its type to $T(f_1) \sqcup T(f_2) \sqcup T(f_3)$ and f_1, f_2 are consumed after performing the operation.

Definition 5.3.4. $\forall f_1, f_2 \in \mathit{types} \ \land \ \forall f_3 \notin \mathit{types}.\ f_1, f_3 \rightarrow^o f_3$ is allowed, provided that f_2 must be assigned the type $T(f_1) \sqcup T(f_2)$ and f_1, f_2 are consumed after performing the operation.

Similarly, copying f_1 to f_2 can be performed while $f_2 \in \mathit{types}$ or $f_2 \notin \mathit{types}$. If $f_2 \notin \mathit{types}$, then it is sufficient to assign to it the type of $\mathit{dst}(T(f_1))$. If $f_2 \in \mathit{types}$, then the type of f_2 must change to $\mathit{dst}(T(f_1)) \sqcup T(f_2)$. Let $f_1 \rightarrow^{copy} f_2$ denotes the flow of information from f_1 to f_2 caused by `copy` operations, then we define the policy for performing `copy` operations as follows.

Definition 5.3.5. $\forall f_1, f_2 \in \mathit{types}$. $f_1 \rightarrow^{copy} f_2$ is allowed if and only if $T(f_1) \in \{\mathrm{UC}, \mathrm{LC}^{n>0}\}$, and f_2 must change its type to $\mathit{dst}(T(f_1)) \sqcup T(f_2)$ and f_1 must change its type to $\mathit{red}(T(f_1))$ after performing the operation.

Definition 5.3.6. $\forall f_1 \in \mathit{types} \wedge \forall f_2 \notin \mathit{types}$. $f_1 \rightarrow^{copy} f_2$ is allowed if and only if $T(f_1) \in \{\mathrm{UC}, \mathrm{LC}^{n>0}\}$, and f_2 must be assigned the type $\mathit{dst}(T(f_1))$ and f_1 must change its type to $\mathit{red}(T(f_1))$ after performing the operation.

To illustrate the above definitions of policies, we give the following examples of operations performed on a set of files with names and types Γ. Based on the definition above, we show which operation is allowed to be performed and which is not, as well as the consequences of performing the operation on the set Γ. Let $\Gamma = \{f_1 : \mathrm{UC}, f_2 : \mathrm{LC}^4, f_3 : \mathrm{LC}^2, f_4 : \mathrm{NC}\}$, then,

$f_1 \rightarrow^o f_3$ is allowed and $\Gamma = \{f_2 : \mathrm{LC}^4, f_3 : \mathrm{LC}^2, f_4 : \mathrm{NC}\}$
$f_3 \rightarrow^o f_1$ is allowed and $\Gamma = \{f_1 : \mathrm{LC}^2, f_2 : \mathrm{LC}^4, f_4 : \mathrm{NC}\}$
$f_4 \rightarrow^o f_5$ is allowed and $\Gamma = \{f_1 : \mathrm{UC}, f_2 : \mathrm{LC}^4, f_3 : \mathrm{LC}^2, f_5 : \mathrm{NC}\}$
$f_3 \rightarrow^o f_2$ is allowed and $\Gamma = \{f_1 : \mathrm{UC}, f_2 : \mathrm{LC}^2, f_4 : \mathrm{NC}\}$
$f_4, f_3 \rightarrow^o f_1$ is allowed and $\Gamma = \{f_1 : \mathrm{NC}, f_2 : \mathrm{LC}^4\}$
$f_4 \rightarrow^{copy} f_2$ is not allowed because $T(f_4) \notin \{\mathrm{UC}, \mathrm{LC}^{n>0}\}$.
$f_3 \rightarrow^{copy} f_1$ is allowed and $\Gamma = \{f_1 : \mathrm{NC}, f_2 : \mathrm{LC}^4, f_3 : \mathrm{LC}^1, f_4 : \mathrm{NC}\}$
$f_1 \rightarrow^{copy} f_5$ is allowed and $\Gamma = \{f_1 : \mathrm{UC}, f_2 : \mathrm{LC}^4, f_3 : \mathrm{LC}^2, f_4 : \mathrm{NC}, f_5 : \mathrm{UC}\}$

In the next section we present the language syntax and semantics.

5.4 Language syntax and semantics

Let $\langle f \rangle$ be a set of valid files names for a given file system. The syntax of the language is given by the following grammar:

$$\begin{array}{lll}
\langle p \rangle & ::= & \langle cs \rangle \mid \langle f \rangle \\
\langle cs \rangle & ::= & \langle c \rangle \mid \langle c \rangle ; \langle cs \rangle \\
\langle c \rangle & ::= & \texttt{cp}\ \langle f \rangle\ \langle f \rangle \mid \texttt{rm}\ \langle f \rangle \mid \texttt{mkf}\ \langle f \rangle\ \langle t \rangle \mid \texttt{rd}\ \langle f \rangle \mid \texttt{cat}\ \langle f \rangle\ \langle f \rangle\ \langle f \rangle \mid \texttt{mv}\ \langle f \rangle\ \langle f \rangle \\
 & \mid & \texttt{copy}\ \langle f \rangle\ \langle f \rangle \mid \texttt{append}\ \langle f \rangle\ \langle f \rangle\ \langle f \rangle \mid \texttt{move}\ \langle f \rangle\ \langle f \rangle \\
\langle t \rangle & ::= & \mathrm{NC} \mid \mathrm{LC}^n \mid \mathrm{UC} \mid \mathrm{void}
\end{array}$$

The language above consists of phrases. A phrase is either a list of commands (cs) or a file name (f). Commands can be either a single command (c) or a sequence of commands (c ; cs). We include commands to copy, remove, make, read, concatenate and move files. These commands operate on a file system δ that we represent as a set of files. A file has a name, content and a type, and we write $f(c) : \tau$ for a file with name f, content c and type τ. For example, $\delta = \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2, \ldots, f_n(c_n) : \tau_n\}$. We use the following notations: $C(f)$ and $T(f)$ denote the content of file f and the type of file f, respectively. $C(f_1) + C(f_2)$ and $T(f_1) \sqcup T(f_2)$ denote concatenating the content of f_1 and f_2, and the join of the types of f_1 and f_2, respectively. We write $\delta[f_2 \leftarrow C(f_1)]$ for updating f_2 with the content of f_1 in the file system δ, and $\delta[f_2 \leftarrow T(f_1)]$ for updating f_2 with the type of f_1 in δ. Both operations require that f_1 and f_2 must exist in δ and $\delta[f_2 \leftarrow C(f_1)]$ requires both files to have distinct names. We write $\delta[-f]$ to remove f from δ if f exists in δ, and $\delta[+f]$ to add f to δ if f does not already exist in δ. We write $\delta[f_3 \leftarrow C(f_1) + C(f_2)]$ for updating f_3 with the concatenated content of f_1 and f_2, and $\delta[f_3 \leftarrow T(f_1) \sqcup T(f_2)]$ for updating f_3 with the join of the types of f_1 and f_2. Both operations require that f_1, f_2 and f_3 must exist in δ and $\delta[f_3 \leftarrow C(f_1) + C(f_2)]$ requires both files to have distinct names. Note that a sequence of operations can be applied to δ in order from left to right. For example, the notation $\delta[+f, -f]$ denotes adding file f first and then removing the file f from δ. We can now put all these ideas together to give the semantics of commands in terms of evaluation rules as shown in Figure 5.2.

We write $\langle e, \delta \rangle \rightarrow \delta'$ for evaluating the command e in δ that yields a new file system δ'. For example, let $\delta = \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2, f_3(c_3) : \tau_3\}$, then

$$\begin{aligned}
&\langle \texttt{cp}\ f_1\ f_2, \delta \rangle \rightarrow \{f_1(c_1) : \mathit{red}(\tau_1), f_2(c_1) : \tau_2 \sqcup \mathit{dst}(\tau_1), f_3(c_3) : \tau_3\}\\
&\langle \texttt{rm}\ f_1, \delta \rangle \rightarrow \{f_2(c_2) : \tau_2, f_3(c_3) : \tau_3\}\\
&\langle \texttt{mkf}\ f_4\ t, \delta \rangle \rightarrow \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2, f_3(c_3) : \tau_3, f_4(c_4) : t\}\\
&\langle \texttt{rd}\ f_1, \delta \rangle \rightarrow \{f_2(c_2) : \tau_2, f_3(c_3) : \tau_3\}\\
&\langle \texttt{cat}\ f_1\ f_2\ f_3, \delta \rangle \rightarrow \{f_3(c_1 + c_2) : \tau_3 \sqcup \tau_1 \sqcup \tau_2\}\\
&\langle \texttt{mv}\ f_1\ f_2, \delta \rangle \rightarrow \{f_2(c_1) : \tau_2 \sqcup \tau_1, f_3(c_3) : \tau_3\}\\
&\langle \texttt{copy}\ f_1\ f_4, \delta \rangle \rightarrow \{f_1(c_1) : \mathit{red}(\tau_1), f_3(c_3) : \tau_3, f_4(c_1) : \mathit{dst}(\tau_1)\}\\
&\langle \texttt{append}\ f_1\ f_2\ f_4, \delta \rangle \rightarrow \{f_3(c_3) : \tau_3, f_4(c_1 + c_2) : \tau_1 \sqcup \tau_2\}\\
&\langle \texttt{move}\ f_1\ f_4, \delta \rangle \rightarrow \{f_2(c_2) : \tau_2, f_3(c_3) : \tau_3, f_4(c_1) : \tau_1\}
\end{aligned}$$

We differentiate between two kinds of commands as shown in the evaluation rules in Figure 5.2. Commands that overwrite existing files and commands that do not overwrite existing files. The commands `cp`, `cat`, and `mv` are those commands which overwrite existing

1. $\langle$`cp` $f_1\ f_2\ ,\delta\ \rangle \rightarrow \delta[f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_2) \sqcup \mathit{dst}(T(f_1))][f_1 \leftarrow \mathit{red}(T(f_1))]$
2. $\langle$`rm` $f, \delta\rangle \rightarrow \delta[-f]$
3. $\langle$`mkf` $f\ t, \delta\rangle \rightarrow \delta[+f][f \leftarrow t]$
4. $\langle$`rd` $f, \delta\rangle \rightarrow \delta[-f]$
5. $\langle$`cat` $f_1\ f_2\ f_3, \delta\rangle \rightarrow \delta[f_3 \leftarrow C(f_1) + C(f_2)][f_3 \leftarrow T(f_1) \sqcup T(f_2) \sqcup T(f_3)][-f_1, -f_2]$
6. $\langle$`mv` $f_1\ f_2\ ,\delta\ \rangle \rightarrow \delta[f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_1) \sqcup T(f_2)][-f_1]$
7. $\langle$`copy` $f_1\ f_2\ ,\delta\ \rangle \rightarrow \delta[+f_2, f_2 \leftarrow C(f_1)][f_2 \leftarrow \mathit{dst}(T(f_1))][f_1 \leftarrow \mathit{red}(T(f_1))]$
8. $\langle$`append` $f_1\ f_2\ f_3, \delta\rangle \rightarrow \delta[+f_3, f_3 \leftarrow C(f_1) + C(f_2)][f_3 \leftarrow T(f_1) \sqcup T(f_2)][-f_1, -f_2]$
9. $\langle$`move` $f_1\ f_2, \delta\ \rangle \rightarrow \delta[+f_2, f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_1)][-f_1]$

Figure 5.2: Single-step semantics

files since they all require that the destination file must exist in δ. On the other hand, the commands `copy`, `append`, and `move` are those commands which do not overwrite existing files since they all require that the destination file must be created first, and thus must not exist in δ. These commands are useful to prevent accidental overwriting of existing files.

The evaluation rules reflect the definitions of policies given in the previous section. The `cp` rule reflects Definition 5.3.5 while the `copy` rule reflects Definition 5.3.6. The `cat` rule reflects Definition 5.3.3 while in the `mv` rule reflects Definition 5.3.1. The `append` rule reflects Definition 5.3.4 while in the `move` rule reflects Definition 5.3.2.

From the single-step transitions, we can define the semantics of sequences of commands in two different ways: we give the small-step semantics in Figure 5.3, and the big-step semantics in Figure 5.4.

For the small-step semantics, we define $\Rightarrow^*$ to be reflexive and transitive closure of $\Rightarrow$ i.e.:

1. if $\langle c, \delta\rangle \Rightarrow \delta'$, then $\langle c, \delta\ \rangle \Rightarrow^* \delta'$.
2. for any $\langle c, \delta\rangle$, $\langle c, \delta\rangle \Rightarrow^* \langle c, \delta\rangle$.
3. if $\langle c, \delta\rangle \Rightarrow^* \delta'$ and $\langle c', \delta'\rangle \Rightarrow^* \delta''$, then $\langle c, \delta\rangle \Rightarrow^* \delta''$

We can now show that these two semantics are equivalent.

$$\frac{\langle c, \delta \ \rangle \Rightarrow \delta'}{\langle c; cs, \delta \rangle \Rightarrow \langle cs, \delta' \rangle} \ (e_{cs})$$

$$\frac{\langle c, \delta \ \rangle \to \delta'}{\langle c, \delta \rangle \Rightarrow \delta'} \ (e_c)$$

Figure 5.3: Small-step semantics

$$\frac{\langle c, \delta \ \rangle \Downarrow \delta' \quad \langle cs, \delta' \ \rangle \Downarrow \delta''}{\langle c; cs, \delta \rangle \Downarrow \delta''} \ (e_{cs})$$

$$\frac{\langle c, \delta \ \rangle \to \delta'}{\langle c, \delta \rangle \Downarrow \delta'} \ (e_c)$$

Figure 5.4: Big-step semantics

Theorem 5.4.1. (Equivalence of semantics) For all commands e, stores δ and δ', we have

$$\text{If } \langle e, \delta \rangle \Downarrow \delta' \iff \langle e, \delta \rangle \Rightarrow^* \delta'$$

Proof. We show each direction separately. If $\langle e, \delta \rangle \Downarrow \delta'$, then $\langle e, \delta \rangle \Rightarrow^* \delta'$. We proceed by induction on the structure of the command e. There are two cases, one for atomic commands and one for the sequence of commands.

1. If e is an atomic command c, then the only rule whose conclusion matches the configuration $\langle c, \delta \rangle \Downarrow \delta'$ is the big-step rule (e_c). By the small-step rule (e_c), we also have $\langle c, \delta \rangle \Rightarrow \delta'$. Thus, we conclude that $\langle c, \delta \rangle \Rightarrow^* \delta'$ as required.

2. If e is a sequence of commands $c; cs$, then the only rule whose conclusion matches the configuration $\langle c; cs, \delta \rangle \Downarrow \delta'$ is the big-step rule (e_{cs}). Since the last rule used in the derivation was (e_{cs}), it must be the case that $\langle c, \delta \rangle \Downarrow \delta'$ and $\langle cs, \delta' \rangle \Downarrow \delta''$ hold as well. By the induction hypothesis twice, we must have $\langle c, \delta \rangle \Rightarrow^* \delta'$, and $\langle cs, \delta' \rangle \Rightarrow^* \delta''$. By the small-step rule (e_{cs}) we have $\langle c; cs, \delta \rangle \Rightarrow \langle cs, \delta' \rangle$. Thus, we conclude that $\langle c; cs, \delta \rangle \Rightarrow^* \delta''$ as required.

We now look at the other direction. If $\langle e, \delta \rangle \Rightarrow^* \delta'$, then $\langle e, \delta \rangle \Downarrow \delta'$. We proceed by induction on the structure of the command e. There are two cases, one for atomic commands and one for the sequence of commands.

1. If e is an atomic command c, then the only rule whose conclusion matches the configuration $\langle c, \delta \rangle \Rightarrow \delta'$ is the small-step rule (e_c). By the big-step rule (e_c), we also have$\langle c, \delta \rangle \Downarrow \delta'$ as required.

2. If e is a sequence of commands $c; cs$, then the only rule whose conclusion matches the configuration $\langle c; cs, \delta \rangle \Rightarrow \langle cs, \delta' \rangle$ is the small-step rule (e_{cs}). Since the last rule used in the derivation was (e_{cs}), it must be the case that $\langle c, \delta \rangle \Rightarrow \delta'$ hold as well. By the induction hypothesis, we must have $\langle c, \delta \rangle \Downarrow \delta'$. By the big-step rule (e_c), we have $\langle c, \delta \rangle \Downarrow \delta'$ as required.

$\square$

In the next section we define the security errors that can occur during evaluation of commands.

5.5 Security errors

A security error is a configuration which fails to evaluate, written as $\langle e, \delta \rangle \rightarrow Err$. Such failure of evaluation results from two kinds of errors which we refer to as *syntactical errors* and *type errors*. In this section we look at each of them independently.

5.5.1 Syntactical errors

Syntactical errors are those errors which occur when the constraints of an operation applied to δ are not satisfied. These constraints are shown in Table 5.1. Evaluating the configuration $\langle e, \delta \rangle$ leads to an error if any of the constraints of an operation applied to δ during the evaluation is not satisfied.

For example, let $\delta = \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2\}$, then $\langle \texttt{mkf}\ f_1\ t, \delta \rangle \rightarrow Err$, because the operation $\delta[+f_1]$ requires f_1 to not exist in δ. Such error prevents accidental overwriting of existing files by creating a file that already exists. $\langle \texttt{rm}\ f_3, \delta \rangle \rightarrow Err$, because the operation $\delta[-f_3]$ requires f_3 to exist in δ. This is a reasonable error since a file needs to exist in order to be removed. $\langle \texttt{mv}\ f_1\ f_1, \delta \rangle \rightarrow Err$, because the operation $\delta[f_1 \leftarrow C(f_1)]$ requires both files to have distinct names. If we allow the configuration $\langle \texttt{mv}\ f_1\ f_1, \delta \rangle$ to evaluate without an error, it will have the same effect as $\langle \texttt{rm}\ f_1, \delta \rangle$, and hence, a file might be removed accidentally. Similarly, $\langle \texttt{cat}\ f_1\ f_2\ f_1, \delta \rangle \rightarrow Err$ for the same reason as the configuration $\langle \texttt{mv}\ f_1\ f_1, \delta \rangle$. $\langle \texttt{append}\ f_1\ f_1\ f_3, \delta \rangle \rightarrow Err$, because the operations $\delta[-f_1, -f_1]$ requires f_1 to exist twice in δ which cannot happen, and also the operation $\delta[f_3 \leftarrow C(f_1) + C(f_1)]$ requires both files to have distinct names.

As shown in Figure 5.2, multiple operations are applied to δ and in order from left to right. In the above examples, we show the first operation that failed in each configuration, which means that previous operations did not fail in the same configuration. For example, in $\langle \texttt{append}\ f_1\ f_1\ f_3, \delta \rangle \rightarrow Err$, we have the following operations applied to δ in order $\delta[+f_3, f_3 \leftarrow C(f_1) + C(f_1)][f_3 \leftarrow T(f_1) \sqcup T(f_1)][-f_1, -f_1]$. However, we showed only the first operation failure, which is $\delta[f_3 \leftarrow C(f_1) + C(f_1)]$, as the operation $[+f_3]$ did not fail because $f_3 \notin \delta$ in the example above.

Operation	Constraints
$\delta[+f]$	$f \notin \delta$
$\delta[-f]$	$f \in \delta$
$\delta[f_2 \leftarrow T(f_1)]$	$f_1, f_2 \in \delta$
$\delta[f_2 \leftarrow C(f_1)]$	$f_1, f_2 \in \delta \wedge f_1 \neq f_2$
$\delta[f_3 \leftarrow T(f_1) \sqcup T(f_2)]$	$f_1, f_2, f_3 \in \delta$
$\delta[f_3 \leftarrow C(f_1) + C(f_2)]$	$f_1, f_2, f_3 \in \delta \wedge f_1 \neq f_2,\ f_1 \neq f_3,\ f_2 \neq f_3$

Table 5.1: Constraints of operations applied to δ

5.5.2 Types errors

Types errors are those errors which occur when the constraints of an operation applied to a type of a file are not satisfied. These constraints are shown in Table 5.2.. Evaluating the configuration $\langle e, \delta \rangle$ leads to an error if any of the constraints of an operation applied to a type during the evaluation is not satisfied.

Operation	Constraints
$\mathit{dst}(\tau)$	$\tau \in \{\mathrm{UC}, \mathrm{LC}^{n>0}\}$
$\mathit{red}(\tau)$	$\tau \in \{\mathrm{UC}, \mathrm{LC}^{n>0}\}$

Table 5.2: Constraints of operations applied to types

For example, let $\delta = \{f_1(c_1) : \mathrm{LC}^0, f_2(c_2) : \mathrm{NC}\}$, then $\langle \texttt{cp}\ f_1\ f_2, \delta \rangle \to \mathit{Err}$, because the operations $\mathit{dst}(\mathrm{LC}^0)$ and $\mathit{red}(\mathrm{LC}^0)$ are applied to LC^0 where $\mathrm{LC}^0 \notin \{\mathrm{UC}, \mathrm{LC}^{n>0}\}$, and $\langle \texttt{copy}\ f_2\ f_3, \delta \rangle \to \mathit{Err}$, because the operations $\mathit{dst}(\mathrm{NC})$ and $\mathit{red}(\mathrm{NC})$ are applied to NC where $\mathrm{NC} \notin \{\mathrm{UC}, \mathrm{LC}^{n>0}\}$.

5.6 Syntactical correctness

The occurrence of file names in a command determines whether or not the command can be evaluated in a particular file system δ without syntactical errors. It should be noted that a guarantee of syntactical-free error of evaluating the command in a file system δ is not a guarantee of being error free. This is because there might be a type error even if there is no syntactical error. In this section we are concerned with syntactical errors and we assume no type errors can occur during evaluation. We write $\langle e, \delta \rangle \to^s \mathit{Err}$ for a configuration that fails to evaluate because of syntactical errors and $\langle e, \delta \rangle \not\rightarrow^s Err$ for a configuration that does not fail because of syntactical errors, and thus, should lead to a new state δ'. Below we discuss the atomic commands and sequence of commands separately, and show when such commands can be evaluated in a state δ without syntactical errors.

5.6.1 Atomic commands

Some atomic commands, such as ($\texttt{rm}\ f$), require the occurrence of file names to exist in δ to be evaluated without syntactical errors, while other commands, such as ($\texttt{mkf}\ f\ t$), require them to not exist in δ. For example, in a file system δ where $\delta = \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2\}$, the configurations $\langle \texttt{rm}\ f_1, \delta \rangle$ and $\langle \texttt{mkf}\ f_3\ t, \delta \rangle$ will not lead to syntactical errors if evaluated, $\langle \texttt{rm}\ f_1, \delta \rangle \not\rightarrow^s Err$ and $\langle \texttt{mkf}\ f_3\ t, \delta \rangle \not\rightarrow^s Err$. This is because the constraints of both commands are satisfied, that is $f_1 \in \delta$ and $f_3 \notin \delta$. On the other hand, the configurations $\langle \texttt{rm}\ f_3, \delta \rangle$ and $\langle \texttt{mkf}\ f_1\ t, \delta \rangle$ will lead to syntactical errors if evaluated, $\langle \texttt{rm}\ f_1, \delta \rangle \rightarrow^s Err$ and $\langle \texttt{mkf}\ f_3\ t, \delta \rangle \rightarrow^s Err$. This is because the constraints of both commands are not satisfied. To determine whether or not an atomic command will lead to a syntactical error if evaluated in a particular file system δ, we need to find out which file names must be in δ and which file names must not be in δ.

Table 5.3 shows the constraints on file names that must be satisfied for each command to be evaluated without syntactical errors. H denotes the set of file names that must exist in δ and N denotes the set of file names that must not exist in δ. We define the function $C(e)$ that takes an atomic command e and returns the set of file names in e that must be in δ and the set of file names in e that must not be in δ, if the file names in e are distinct from each other. Therefore, we write $C(e) = (H, N)$ if and only if the command e satisfies the condition in the table, where (H, N) are the sets of file names of the command e as shown in the table. For example, $C(\texttt{cp}\ f_1\ f_2) = (\{f_1, f_2\}, \varnothing)$, since $f_1 \neq f_2$. However, $C(\texttt{cp}\ f_1\ f_1)$ should fail, since it is not the case that $f_1 \neq f_1$. Below we give a proof of that a configuration $\langle e, \delta \rangle$ will not lead to a syntactical error if evaluated, $\langle e, \delta \rangle \not\rightarrow^s Err$, if $C(e) = (H, N)$ and $H \subseteq \delta$ and $N \cap \delta = \varnothing$.

Theorem 5.6.1. *If $C(e) = (H, N)$ and $H \subseteq \delta$ and $N \cap \delta = \varnothing$, then $\langle e, \delta \rangle \not\rightarrow^s Err$.*

Proof. We proceed by cases on the atomic commands e. There are 9 cases, here we show a selection of them.

1. If e is the command $\texttt{cp}\ f_1\ f_2$, then we have $C(\texttt{cp}\ f_1\ f_2) = (\{f_1, f_2\}, \varnothing)$ and $\{f_1, f_2\} \subseteq \delta$ and $\varnothing \cap \delta = \varnothing$. Now we can apply rule (1) to obtain $\langle \texttt{cp}\ f_1\ f_2\ , \delta\ \rangle \rightarrow \delta[f_2 \leftarrow$

Commands	H	N	Condition
`cp` f_1 f_2	$\{f_1, f_2\}$	$\varnothing$	$f_1 \neq f_2$
`rm` f	$\{f_1\}$	$\varnothing$	–
`mkf` f t	$\varnothing$	$\{f_1\}$	–
`rd` f	$\{f_1\}$	$\varnothing$	–
`cat` f_1 f_2 f_3	$\{f_1, f_2, f_3\}$	$\varnothing$	$f_1 \neq f_2, f_1 \neq f_3, f_2 \neq f_3$
`mv` f_1 f_2	$\{f_1, f_2\}$	$\varnothing$	$f_1 \neq f_2$
`copy` f_1 f_2	$\{f_1\}$	$\{f_2\}$	$f_1 \neq f_2$
`append` f_1 f_2 f_3	$\{f_1, f_2\}$	$\{f_3\}$	$f_1 \neq f_2, f_1 \neq f_3, f_2 \neq f_3$
`move` f_1 f_2	$\{f_1\}$	$\{f_2\}$	$f_1 \neq f_2$

Table 5.3: Constraints for atomic commands

$C(f_1)][f_2 \leftarrow T(f_2) \sqcup \mathit{dst}(T(f_1))][f_1 \leftarrow \mathit{red}(T(f_1))]$. Since the operations $\delta[f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_2) \sqcup \mathit{dst}(T(f_1))][f_1 \leftarrow \mathit{red}(T(f_1))]$ require $f_1 \in \delta$ and $f_2 \in \delta$, and we have $\{f_1, f_2\} \subseteq \delta$, then $\langle \texttt{cp}\ f_1\ f_2, \delta\rangle \not\rightarrow^s \mathit{Err}$ as required.

2. If e is the command `rm` f, then we have $C(\texttt{rm}\ f) = (\{f\}, \varnothing)$ and $\{f\} \subseteq \delta$ and $\varnothing \cap \delta = \varnothing$. Now we can apply rule (2) to obtain $\langle \texttt{rm}\ f, \delta\rangle \rightarrow \delta[-f]$. Since the operation $\delta[-f]$ requires $f \in \delta$, and we have $\{f\} \subseteq \delta$, then $\langle \texttt{rm}\ f, \delta\rangle \not\rightarrow^s \mathit{Err}$.

3. If e is the command `mkf` f τ, then we have $C(\texttt{mkf}\ f\ \tau) = (\varnothing, \{f\})$ and $\varnothing \subseteq \delta$ and $\{f\} \cap \delta = \varnothing$. Now we can apply rule (3) to obtain $\langle \texttt{mkf}\ f\ t, \delta\rangle \rightarrow \delta[+f][f \leftarrow t]$. Since the operation $\delta[+f]$ requires $f \notin \delta$ and we have $\{f\} \cap \delta = \varnothing$, then the operation can be successfully applied to δ. Also, since the operation $[f \leftarrow t]$ requires $f \in \delta$, and we have established that $\delta[+f]$ can be applied to δ successfully, it must be case that $f \in \delta$ after applying the operation $\delta[+f]$, therefore, $\langle \texttt{mkf}\ f\ \tau, \delta\rangle \not\rightarrow^s \mathit{Err}$.

4. If e is the command `rd` f, then we have $C(\texttt{rd}\ f) = (\{f\}, \varnothing)$ and $\{f\} \subseteq \delta$ and $\varnothing \cap \delta = \varnothing$. Now we can apply rule (4) to obtain $\langle \texttt{rd}\ f, \delta\rangle \rightarrow \delta[-f]$. Since the operation $\delta[-f]$ requires $f \in \delta$, and we have $\{f\} \subseteq \delta$, then $\langle \texttt{rd}\ f, \delta\rangle \not\rightarrow^s \mathit{Err}$.

5. If e is the command `copy` f_1 f_2, then we have $C(\texttt{copy}\ f_1\ f_2) = (\{f_1\}, \{f_2\})$ and $\{f_1\} \subseteq \delta$ and $\{f_2\} \cap \delta = \varnothing$. Now we can apply rule (7) to obtain $\langle \texttt{copy}\ f_1\ f_2\ , \delta\ \rangle \rightarrow \delta[+f_2, f_2 \leftarrow C(f_1)][f_2 \leftarrow \mathit{dst}(T(f_1))][f_1 \leftarrow \mathit{red}(T(f_1))]$. Since the operation $\delta[+f_2]$ requires $f_2 \notin \delta$, and we have $\{f_2\} \cap \delta = \varnothing$, then the operation can be successfully applied to δ. Also, since the operations $\delta[f_2 \leftarrow C(f_1)][f_2 \leftarrow \mathit{dst}(T(f_1))][f_1 \leftarrow$

$red(T(f_1))$] require $f_1 \in \delta$ and $f_2 \in \delta$, and we have $\{f_1\} \subseteq \delta$ and we have established that the operation $\delta[+f_2]$ can be applied to δ successfully, it must be the case that $f_2 \in \delta$ after applying the operation, therefore, $\langle \texttt{copy}\ f_1\ f_2\ , \delta\ \rangle \not\to^s \mathit{Err}$.

□

5.6.2 Sequence of commands

Theorem 5.6.1 illustrates the constraints for an atomic command to be evaluated in a file system δ without syntactical errors. However, applying these constraints to commands individually to determine whether or not a sequence of commands can be evaluated in a file system δ without syntactical errors does not work. In other words, even if atomic commands can be evaluated individually in a file system δ without syntactical errors, evaluating them in a sequence in δ might lead to syntactical errors. For example, let $\delta = \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2\}$. Then, the configuration $\langle \texttt{rm}\ f_1; \texttt{rm}\ f_1, \delta\rangle \not\to^s \mathit{Err}$, because $\langle \texttt{rm}\ f_1, \delta\rangle \not\to^s \mathit{Err}$ and $\langle \texttt{rm}\ f_1, \delta\rangle \not\to^s \mathit{Errr}$. This is because for both configurations the constraints of the command are satisfied, that is $C(\texttt{rm}\ f_1) = (\{f_1\}, \varnothing)$ and $\{f_1\} \subseteq \delta$ and $\varnothing \cap \delta = \varnothing$. However, by applying the small-step (e_{cs}) rule to the configuration $\langle \texttt{rm}\ f_1; \texttt{rm}\ f_1, \delta\rangle$ we have $\langle \texttt{rm}\ f_1; \texttt{rm}\ f_1, \delta\rangle \Rightarrow \langle \texttt{rm}\ f_1, \delta'\rangle$ where $\delta' = \{f_2(c_2) : \tau_2\}$ as evaluating the first command removes f_1 from δ. Now by applying the small-step (e_c) rule to the configuration $\langle \texttt{rm}\ f_1, \delta'\rangle$ we have $\langle \texttt{rm}\ f_1, \delta'\rangle \Rightarrow^s \mathit{Err}$ because $f_1 \notin \delta'$. Similarly, $\langle \texttt{mkf}\ f_3\ t; \texttt{mkf}\ f_3\ t, \delta\rangle \not\to^s \mathit{Errr}$, because $\langle \texttt{mkf}\ f_3\ t, \delta\rangle \not\to^s \mathit{Errr}$ and $\langle \texttt{mkf}\ f_3\ t, \delta\rangle \not\to^s \mathit{Errr}$. This is because for both configurations the constraints of the command are satisfied, that is $C(\texttt{mkf}\ f_3\ t) = (\varnothing, \{f_3\})$ and $\varnothing \subseteq \delta$ and $\{f_3\} \cap \delta = \varnothing$. However, by applying the small-step (e_{cs}) rule to the configuration $\langle \texttt{mkf}\ f_3\ t; \texttt{mkf}\ f_3\ t, \delta\rangle$ we have $\langle \texttt{mkf}\ f_3\ t; \texttt{mkf}\ f_3\ t, \delta\rangle \Rightarrow \langle \texttt{mkf}\ f_3\ t, \delta'\rangle$ where $\delta = \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2, f_3(c_3) : \tau_3\}$ as evaluating the first command creates f_3 in δ. Now by applying the small-step (e_c) rule to the configuration $\langle \texttt{mkf}\ f_3\ t, \delta'\rangle$, we have $\langle \texttt{mkf}\ f_3\ t, \delta'\rangle \Rightarrow^s \mathit{Err}$, because $f_3 \in \delta'$.

One the other hand, even if atomic commands will lead to a syntactical error if evaluated individually in a file system δ, evaluating them in a sequence in δ might not lead to syntactical errors. For example, the configuration $\langle \texttt{rm}\ f_1; \texttt{mkf}\ f_1\ t, \delta\rangle \to^s \mathit{Err}$, because $\langle \texttt{mkf}\ f_1\ t, \delta\rangle \to^s \mathit{Err}$. This because the constraints of the command in the configuration $\langle \texttt{mkf}\ f_1\ t, \delta\rangle$ are not satisfied, that is $C(\texttt{mkf}\ f_1\ t) = (\varnothing, \{f_1\})$ and $\varnothing \subseteq \delta$ and $\{f_1\} \cap \delta \neq \varnothing$. However, by applying the small-step (e_{cs}) rule to the configuration $\langle \texttt{rm}\ f_1; \texttt{mkf}\ f_1\ t, \delta\rangle$ we have $\langle \texttt{rm}\ f_1; \texttt{mkf}\ f_1\ t, \delta\rangle \Rightarrow \langle \texttt{mkf}\ f_1\ t, \delta'\rangle$ where $\delta' = \{f_2(c_2) : \tau_2\}$ as evaluating the first command removes f_1 from δ. Now by applying the small-step (e_c) rule to the configuration

$\langle \texttt{mkf}\ f_1\ t, \delta' \rangle$, we have $\langle \texttt{mkf}\ f_1\ t, \delta' \rangle \Rightarrow \delta''$, where $\delta'' = \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2\}$ as evaluating the second command creates f_1 in δ. Similarly, $\langle \texttt{mkf}\ f_3\ t; \texttt{rm}\ f_3, \delta \rangle \rightarrow^s Err$, because $\langle \texttt{rm}\ f_3, \delta \rangle \rightarrow^s Err$. This is because the constraints of the command in the configuration $\langle \texttt{rm}\ f_3, \delta \rangle$ are not satisfied, that is $C(\texttt{rm}\ f_3) = (\{f_3\}, \varnothing)$ and $\{f_3\} \nsubseteq \delta$ and $\varnothing \cap \delta = \varnothing$. However, by applying the small-step (e_{cs}) rule to the configuration $\langle \texttt{mkf}\ f_3\ t; \texttt{rm}\ f_3, \delta \rangle$ we have $\langle \texttt{mkf}\ f_3\ t; \texttt{rm}\ f_3, \delta \rangle \Rightarrow \langle \texttt{rm}\ f_3, \delta' \rangle$ where $\delta' = \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2, f_3(c_3) : \tau_3\}$ as evaluating the first command creates f_3 in δ. Now by applying the small-step (e_c) rule to the configuration $\langle \texttt{rm}\ f_1, \delta' \rangle$, we have $\langle \texttt{rm}\ f_1, \delta' \rangle \Rightarrow \delta''$, where $\delta'' = \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2\}$ as evaluating the second command removes f_3 from δ'.

Since commands evaluation changes the state δ, such changes must be considered by subsequent commands when evaluated. Some evaluation of commands remove file names from δ, therefore, such file names can be created but not removed or used by subsequent commands. For example, $\langle \texttt{rm}\ f_1; \texttt{mkf}\ f_1\ s\tau, \delta \rangle \not\rightarrow^s Err$ but $\langle \texttt{rm}\ f_1; \texttt{rm}\ f_1, \delta \rangle \rightarrow^s Err$ and $\langle \texttt{rm}\ f_1; \texttt{cp}\ f_1\ f_2, \delta \rangle \rightarrow^s Err$. Other evaluations of commands create file names in δ, therefore, such file names can be removed or used but not created by subsequent commands. For example $\langle \texttt{mkf}\ f_3\ s\tau; \texttt{rm}\ f_3, \delta \rangle \not\rightarrow^s Err$ and $\langle \texttt{mkf}\ f_3\ s\tau; \texttt{cp}\ f_3\ f_2, \delta \rangle \not\rightarrow^s Err$ but $\langle \texttt{mkf}\ f_3\ s\tau; \texttt{mkf}\ f_3\ s\tau, \delta \rangle \rightarrow^s Err$.

To find out whether a sequence of commands can be evaluated in a file system δ without syntactical errors, we must first find out whether the sequence of commands is consistent or not. A sequence of commands is consistent if and only if each command in the sequence satisfies the following two conditions. Firstly, any file name of a command that needs to be in δ must not have been removed by a previous command. Secondly, any file name of a command that needs not be in δ must not have been created by a previous command. Table 5.4 shows the set of files that are removed or created by evaluating each command. C denotes the set of files that are created by evaluating the command and E denotes the set of files that are erased by evaluating the command.

We define an algorithm that given a sequence of commands, the algorithm succeeds if the commands are consistent. Additionally, the algorithm finds the minimum set of file names that must be in δ and the minimum set of file names that must not be in δ for the sequence of commands to be evaluated without syntactical errors. For a given command cs, we compute 4-tuple (H, N, C, E) that gives the constraints on a starting file system δ so that it can be evaluated without syntactical errors. H denotes the set of file names that must exist in δ, N denotes the set of file names that must not exist in δ. C denotes

Commands	C	E
`cp` f_1 f_2	$\varnothing$	$\varnothing$
`rm` f	$\varnothing$	$\{f\}$
`mkf` f t	$\{f\}$	$\varnothing$
`rd` f	$\varnothing$	$\{f_1\}$
`cat` f_1 f_2 f_3	$\varnothing$	$\{f_1, f_2\}$
`mv` f_1 f_2	$\varnothing$	$\{f_1\}$
`copy` f_1 f_2	$\{f_2\}$	$\varnothing$
`append` f_1 f_2 f_3	$\{f_3\}$	$\{f_1, f_2\}$
`move` f_1 f_2	$\{f_2\}$	$\{f_1\}$

Table 5.4: File creation and erasure by commands

the set of file names that are created by the sequence of commands, such file names do not necessarily have to be free in δ initially. E denotes the set of file names that are erased by the sequence of commands, such file names do not necessarily have to be in δ initially. Table 5.5 gives the heart of the algorithm. We write $c(H, N, C, E) = (H', N', C', E')$ if an atomic command c satisfies the conditions in the table, where (H', N', C', E') are the sets updated by the command c. The algorithm starts with $(\varnothing, \varnothing, \varnothing, \varnothing)$. For example, `cp` f_1 $f_2(\varnothing, \varnothing, \varnothing, \varnothing) = (\{f_1, f_2\}, \varnothing, \varnothing, \varnothing)$. This means that the files $\{f_1, f_2\}$ must be part of the file system when this command is evaluated. When a command does not satisfy the conditions in the table, the algorithm fails. For example, `cp` f_1 $f_1(\varnothing, \varnothing, \varnothing, \varnothing)$ should fails since the condition $f_1 \neq f_1$ is not satisfied. Sequences of commands are then computed by composition: $c; cs(\varnothing, \varnothing, \varnothing, \varnothing) = cs(c(\varnothing, \varnothing, \varnothing, \varnothing))$. Below we give several examples to show how the algorithm works for sequences of commands.

Example 5.6.1. `rm` f_1; `rm` $f_1(\varnothing, \varnothing, \varnothing, \varnothing)$

$$\texttt{rm}\ f_1(\texttt{rm}\ f_1(\varnothing, \varnothing, \varnothing, \varnothing)) = \texttt{rm}\ f_1(\{f_1\}, \varnothing, \varnothing, \{f_1\})\ \textit{since}\ f_1 \notin E$$

$$\texttt{rm}\ f_1(\{f_1\}, \varnothing, \varnothing, \{f_1\}) =\ \textit{fails since}\ f_1 \in E$$

Example 5.6.2. `mkf` f_1 t; `mkf` f_1 $t(\varnothing, \varnothing, \varnothing, \varnothing)$

$$\texttt{mkf}\ f_1\ t(\texttt{mkf}\ f_1\ t(\varnothing, \varnothing, \varnothing, \varnothing)) = \texttt{mkf}\ f_1\ t(\varnothing, \{f_1\}, \{f_1\}, \varnothing)\ \textit{since}\ f_1 \notin C$$

$$\texttt{mkf}\ f_1\ t(\varnothing, \{f_1\}, \{f_1\}, \varnothing) =\ \textit{fails since}\ f_1 \in C$$

Term	**H**	**N**	**C**	**E**	**Condition**
`cp` f_1 f_2	$H \cup (\{f_1, f_2\} - C)$	N	C	E	$f_1 \notin E, f_2 \notin E,$ $f_1 \neq f_2$
`rm` f	$H \cup (\{f\} - C)$	N	$C - \{f\}$	$E \cup \{f\}$	$f \notin E$
`mkf` f t	H	$N \cup (\{f\} - E)$	$C \cup \{f\}$	$E - \{f\}$	$f \notin C$
`rd` f	$H \cup (\{f\} - C)$	N	$C - \{f\}$	$E \cup \{f\}$	$f \notin E$
`cat` f_1 f_2 f_3	$H \cup (\{f_1, f_2, f_3\} - C)$	N	$C - \{f_1, f_2\}$	$E \cup (\{f_1, f_2\}$	$f_1 \notin E, f_2 \notin E,$ $f_3 \notin E, f_1 \neq f_2,$ $f_1 \neq f_3, f_2 \neq f_3$
`mv` f_1 f_2	$H \cup (\{f_1, f_2\} - C)$	N	$C - \{f_1\}$	$E \cup \{f_1\}$	$f_1 \notin E, f_2 \notin E,$ $f_1 \neq f_2$
`copy` f_1 f_2	$H \cup (\{f_1\} - C)$	$N \cup (\{f_2\} - E)$	$C \cup \{f_2\}$	$E - \{f_2\}$	$f_1 \notin E, f_2 \notin C$ $f_1 \neq f_2$
`append` f_1 f_2 f_3	$H \cup (\{f_1, f_2\} - C)$	$N \cup (\{f_3\} - E)$	$(C \cup \{f_3\}) - \{f_1, f_2\}$	$(E \cup (\{f_1, f_2\}) - \{f_3\}$	$f_1 \notin E, f_2 \notin E,$ $f_3 \notin C, f_1 \neq f_2,$ $f_1 \neq f_3, f_2 \neq f_3$
`move` f_1 f_2	$H \cup (\{f_1\} - C)$	$N \cup (\{f_2\} - E)$	$(C \cup \{f_2\}) - \{f_1\}$	$(E \cup (\{f_1\}) - \{f_2\}$	$f_1 \notin E, f_2 \notin C$ $f_1 \neq f_2$

Table 5.5: Constraints for sequence of commands

Example 5.6.3. $\texttt{mkf}\ f_1\ t; \texttt{rm}\ f_1; \texttt{mkf}\ f_1\ t(\emptyset, \emptyset, \emptyset, \emptyset)$

$$\texttt{rm}\ f_1; \texttt{mkf}\ f_1\ t(\texttt{mkf}\ f_1\ t(\emptyset, \emptyset, \emptyset, \emptyset)) = \texttt{rm}\ f_1; \texttt{mkf}\ f_1\ t(\emptyset, \{f_1\}, \{f_1\}, \emptyset)\ \textit{since}\ f_1 \notin C$$

$$\texttt{mkf}\ f_1\ t(\texttt{rm}\ f_1(\emptyset, \{f_1\}, \{f_1\}, \emptyset)) = \texttt{mkf}\ f_1\ t(\emptyset, \{f_1\}, \emptyset, \{f_1\})\ \textit{since}\ f_1 \notin E$$

$$\texttt{mkf}\ f_1\ t(\emptyset, \{f_1\}, \emptyset, \{f_1\}) = (\emptyset, \{f_1\}, \{f_1\}, \emptyset)\ \textit{since}\ f_1 \notin C$$

Example 5.6.4. $\texttt{rm}\ f_1; \texttt{mkf}\ f_1\ t; \texttt{rm}\ f_1(\emptyset, \emptyset, \emptyset, \emptyset)$

$$\texttt{mkf}\ f_1\ t; \texttt{rm}\ f_1(\texttt{rm}\ f_1(\emptyset, \emptyset, \emptyset, \emptyset)) = \texttt{mkf}\ f_1\ t; \texttt{rm}\ f_1(\{f_1\}, \emptyset, \emptyset, \{f_1\})\ \textit{since}\ f_1 \notin E$$

$$\texttt{rm}\ f_1(\texttt{mkf}\ f_1\ t(\{f_1\}, \emptyset, \emptyset, \{f_1\})) = \texttt{rm}\ f_1(\{f_1\}, \emptyset, \{f_1\}, \emptyset)\ \textit{since}\ f_1 \notin C$$

$$\texttt{rm}\ f_1(\{f_1\}, \emptyset, \{f_1\}, \emptyset) = (\{f_1\}, \emptyset, \emptyset, , \{f_1\})\ \textit{since} f_1 \notin E$$

Example 5.6.5. $\texttt{mkf}\ f_2\ t; \texttt{move}\ f_1\ f_2(\emptyset, \emptyset, \emptyset, \emptyset)$

$$\texttt{move}\ f_1\ f_2(\texttt{mkf}\ f_2\ t(\emptyset, \emptyset, \emptyset, \emptyset)) = \texttt{move}\ f_1\ f_2(\emptyset, \{f_2\}, \{f_2\}, \emptyset)\ \textit{since}\ f_2 \notin C$$

$$\texttt{move}\ f_1\ f_2(\emptyset, \{f_2\}, \{f_2\}, \emptyset) = \ fails\ since\ f_2 \in C$$

We can relate these syntactical constraints with the operational semantics through the following result which states that if the file system satisfies the constraints needed for a command as set out above, then it will be evaluated without syntactical errors. Essentially, this key result gives the constraints on the file system: which files must be present, and which files must not be present.

Theorem 5.6.2. For any command sequence cs, if $cs(\varnothing, \varnothing, \varnothing, \varnothing) = (H, N, C, E)$, then for any file system δ, if $H \subseteq \delta$ and $N \cap \delta = \varnothing$, then $\langle cs, \delta \rangle \Rightarrow^* \delta'$.

Proof. We prove a stronger result. For any command sequence cs, if

$$cs(H, N, C, E) = (H', N', C', E')$$

then for any file system δ, if

$$(H' \cup C) - E \subseteq \delta \text{ and } (N' - C) \cup E \cap \delta = \varnothing$$

then

1. $\langle cs, \delta \rangle \Rightarrow^* \delta'$, and
2. $(H' \cup C') - E' \subseteq \delta'$
3. $(N' - C') \cup E' \cap \delta' = \varnothing$.

There are 10 cases, here we show a selection of them.

1. If c is $\texttt{rm}\ f$, then assume

   $$\texttt{rm}\ f(H, N, C, E) = (H \cup (\{f\} - C), N, C - \{f\}, E \cup \{f\})$$

   succeeds, and therefore $f \notin E$, and $(H \cup (\{f\} - C) \cup C) - E \subseteq \delta$ and $(N - C) \cup E \cap \delta = \varnothing$.

   Now, to show that $\langle \texttt{rm}\ f, \delta \rangle \rightarrow \delta[-f]$, we need to show that $f \in \delta$. We shall show that $f \in (H \cup (\{f\} - C) \cup C) - E$.

   First, note that $f \notin E$, so we can simplify the problem to check:

   $f \in (H \cup (\{f\} - C) \cup C)$.

   There are two cases to consider:

   Either $f \in C$: $(H \cup C)$, then $f \in \delta$.

   or $f \notin C$: $(H \cup (\{f\} - C) \cup C) = (H \cup \{f\} \cup C)$, and again $f \in \delta$.

   Therefore, $f \in \delta$, and $(N - C) \cup E \cap \delta = \varnothing$, so the command succeeds.

Now, to show that $(H \cup (C - \{f\}) - (E \cup \{f\})) \subseteq \delta[-f]$, we need to show that $f \notin (H \cup (C - \{f\}) - (E \cup \{f\}))$, which follows by set-theoretical arguments. Therefore, $(H \cup (C - \{f\}) - (E \cup \{f\})) \subseteq \delta[-f]$.

Now, we need to show that $(N - (C - \{f\})) \cup (E \cup \{f\}) \cap \delta[-f] = \varnothing$. This is true by assumption(the $[-f]$ does not change the result).

2. if c is `mkf` f, then assume

$$\texttt{mkf}\ f(H,N,C,E) = (H, N \cup (\{f\} - E), C \cup \{f\}, E - \{f\})$$

succeeds, and therefore $f \notin C$, and $(H \cup C) - E \subseteq \delta$ and $((N \cup (\{f\} - E)) - C) \cup E \cap \delta = \varnothing$.

Now, to show that $\langle \texttt{mkf}\ f, \delta\rangle \to \delta[+f]$, we need to show that $f \notin \delta$. We shall show that $f \in ((N \cup (\{f\} - E)) - C) \cup E$.

First, note that $f \notin C$, so we can simplify the problem to check:

$f \in (N \cup (\{f\} - E)) \cup E$

There are two cases to consider:

Either $f \notin E : (N \cup \{f\} \cup E)$, then $f \notin \delta$.

or $f \in E : (N \cup E)$, then again $f \notin \delta$.

Therefore, $f \notin \delta$, and $(H \cup C) - E \subseteq \delta$, so the command succeeds.

Now, to show that $(H \cup (C \cup \{f\})) - (E - \{f\}) \subseteq \delta[+f]$, we need to show that $f \in (H \cup (C \cup \{f\})) - (E - \{f\})$, which follows by set-theoretical arguments. Therefore, $(H \cup (C \cup \{f\})) - (E - \{f\}) \subseteq \delta[+f]$.

Now, to show that $((N \cup (\{f\} - E)) - (C \cup \{f\})) \cup (E - \{f\}) \cap \delta[+f] = \varnothing$, we need to show that $f \notin ((N \cup (\{f\} - E)) - (C \cup \{f\})) \cup (E - \{f\})$, which follows by set-theoretical arguments.

Therefore, $(N \cup (\{f\} - E) - (C \cup \{f\}) \cup E - \{f\}) \cap \delta[+f] = \varnothing$.

3. if c is `copy` f_1 f_2, then assume

$$\texttt{copy}\ f_1\ f_2(H,N,C,E) = (H \cup (\{f_1\} - C), N \cup (\{f_2\} - E), C \cup \{f_2\}, E - \{f_2\})$$

succeeds, and therefore $f_1 \notin E$, $f_2 \notin C$, and $f_1 \neq f_2$, $((H \cup (\{f_1\} - C)) \cup C) - E \subseteq \delta$ and $((N \cup (\{f_2\} - E)) - C) \cup E \cap \delta = \varnothing$.

Now, to show that $\langle \texttt{copy}\ f_1\ f_2\ , \delta\ \rangle \rightarrow \delta[+f_2, f_2 \leftarrow C(f_1)][f_2 \leftarrow \mathit{dst}(T(f_1))][f_1 \leftarrow \mathit{red}(T(f_1))]$, we need to show that $f_1 \in \delta$ and $f_2 \notin \delta$. We shall sow that $f_1 \in ((H \cup (\{f_1\} - C)) \cup C) - E$ and $f_2 \in ((N \cup (\{f_2\} - E)) - C) \cup E$.

Case $f_1 \in ((H \cup (\{f_1\} - C)) \cup C) - E$:

First, note that $f_1 \notin E$, so we can simplify the problem to check:

$f_1 \in ((H \cup (\{f_1\} - C)) \cup C)$.

There are two cases to consider:

Either $f_1 \in C$: $(H \cup C)$, then $f \in \delta$.

or $f_1 \notin C$: $(H \cup (\{f_1\} - C) \cup C) = (H \cup \{f_1\} \cup C)$, and again $f_1 \in \delta$.

Case $f_2 \in ((N \cup (\{f_2\} - E)) - C) \cup E$.

First, note that $f_2 \notin C$, so we can simplify the problem to check:

$(N \cup (\{f_2\} - E)) \cup E$.

There are two cases to consider:

Either $f_2 \notin E : (N \cup \{f\} \cup E)$, then $f_2 \notin \delta$.

or $f_2 \in E : (N \cup E)$, then again $f_2 \notin \delta$.

Therefore, $f_1 \in \delta$ and $f_2 \notin \delta$, so the command succeeds.

Now, to show that $((H \cup (\{f_1\} - C)) \cup (C \cup \{f_2\})) - (E - \{f_2\}) \subseteq \delta'$, we need to show that $f_1, f_2 \in ((H \cup (\{f_1\} - C)) \cup (C \cup \{f_2\})) - (E - \{f_2\})$, which follows by set-theoretical arguments.

Therefore, $((H \cup (\{f_1\} - C)) \cup (C \cup \{f_2\})) - (E - \{f_2\}) \subseteq \delta'$.

Now to show that $((N \cup (\{f_2\} - E)) - (C \cup \{f_2\})) \cup (E - \{f_2\}) \cap \delta' = \varnothing$, we need to show that $f_2 \notin ((N \cup (\{f_2\} - E)) - (C \cup \{f_2\})) \cup (E - \{f_2\})$, which follows by set-theoretical arguments.

Therefore, $((N \cup (\{f_2\} - E)) - (C \cup \{f_2\})) \cup (E - \{f_2\}) \cap \delta' = \varnothing$.

$\square$

5.7 Type system

In the previous section we presented the constraints that must be satisfied by commands in order to be evaluated in a file system δ without syntactical errors. However, even in the absence of syntactical errors, there might be a type error that leads a configuration to fail to evaluate. In this section, we develop a type system that determines whether or not commands can be evaluated in a file system δ without type errors as well as without syntactical errors. Typing judgements have the form

$$\Gamma \mid \Gamma' \vdash p : \tau$$

where Γ is a set of files with names and types of the form $f : \tau$. We write $\varnothing$ for the empty set. For example, $\Gamma = \{f_1 : \tau_1,\ f_2 : \tau_2,\ f_3 : \tau_3, \ldots, f_n : \tau_n\}$. It should be noted that files in the context Γ are unique and the symbol "," is the disjoint union operation, so that the set of files in Γ does not contain repetitions. The judgement $\Gamma \mid \Gamma' \vdash p : \tau$ means that typing the phrase p of type τ in the context Γ, will change the context to Γ'. In other words, the contexts Γ and Γ' represent the set of files before and after typing the phrase p. Note that a phrase p could be a command, a file name, or a sequence of commands. The typing rules are shown in Figure 5.5. In the next sections we present the typing rule for each phrase individually and give examples to show which phrase is typable and which is not. A phrase is typable if there exists a derivation for it, otherwise is not typable.

5.7.1 Typing rule for file names

$$\frac{}{\Gamma, f : \tau \mid \Gamma \vdash f : \tau}\ (f)$$

The typing rule for a file name f says that typing a file from the context Γ consumes the file from the context, provided that $f \in \Gamma$.

Example 5.7.1. The file f_1 is typable in the context $\Gamma = \{f_1 : \mathrm{NC}\}$ since $f_1 \in \Gamma$ as shown below.

$$\frac{}{f_1 : \mathrm{NC} \mid \varnothing \vdash f_1 : \mathrm{NC}}\ (f)$$

Example 5.7.2. The file f_2 is not typable in the context $\Gamma = \{f_1 : \mathrm{NC}\}$ since $f_2 \notin \Gamma$ as shown below.

$$\frac{}{f_1 : \mathrm{NC} \mid ? \vdash f_2 : ?}\ (?)$$

5.7.2 Typing rule for `cp` command

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \tau \sqsubseteq \mathrm{LC}^{n>0} \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau'}{\Gamma \mid \Gamma'', f_1 : \mathit{red}(\tau), f_2 : \tau' \sqcup \mathit{dst}(\tau) \vdash \texttt{cp}\ f_1\ f_2 : \mathrm{void}}\ (\texttt{cp})$$

The typing rule for `cp` command says that if we can type f_1 and f_2 from the context Γ and f_1 is of type UC or $\mathrm{LC}^{n>0}$, then we can type the command `cp` f_1 f_2 of type void and the type of f_2 is changed to be the least upper bound of its type and the type of $\mathit{dst}(T(f_2))$, and the type of f_1 is changed to be $\mathit{red}(T(f_1))$.

$$\frac{}{\Gamma, f : \tau \mid \Gamma \vdash f : \tau}\,(f) \qquad \frac{\Gamma \mid \Gamma' \vdash c : \text{void} \quad \Gamma' \mid \Gamma'' \vdash cs : \text{void}}{\Gamma \mid \Gamma'' \vdash c; cs : \text{void}}\,(cs)$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \tau \sqsubseteq \text{LC}^{n>0} \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau'}{\Gamma \mid \Gamma'', f_1 : \mathit{red}(\tau), f_2 : \tau' \sqcup \mathit{dst}(\tau) \vdash \texttt{cp}\ f_1\ f_2 : \text{void}}\,(\texttt{cp})$$

$$\frac{\Gamma \mid \Gamma' \vdash f : \tau}{\Gamma \mid \Gamma' \vdash \texttt{rm}\ f : \text{void}}\,(\texttt{rm}) \qquad \frac{}{\Gamma \mid \Gamma, f : t \vdash \texttt{mkf}\ f\ t : \text{void}}\,(\texttt{mkf}) \qquad \frac{\Gamma \mid \Gamma' \vdash f : \tau}{\Gamma \mid \Gamma' \vdash \texttt{rd}\ f : \text{void}}\,(\texttt{rd})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau' \quad \Gamma'' \mid \Gamma''' \vdash f_3 : \tau''}{\Gamma \mid \Gamma''', f_3 : \tau \sqcup \tau' \sqcup \tau'' \vdash \texttt{cat}\ f_1\ f_2\ f_3 : \text{void}}\,(\texttt{cat})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau'}{\Gamma \mid \Gamma'', f_2 : \tau \sqcup \tau' \vdash \texttt{mv}\ f_1\ f_2 : \text{void}}\,(\texttt{mv})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \tau \sqsubseteq \text{LC}^{n>0}}{\Gamma \mid \Gamma', f_1 : \mathit{red}(\tau), f_2 : \mathit{dst}(\tau) \vdash \texttt{copy}\ f_1\ f_2 : \text{void}}\,(\texttt{copy})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau'}{\Gamma \mid \Gamma'', f_3 : \tau \sqcup \tau' \vdash \texttt{append}\ f_1\ f_2\ f_3 : \text{void}}\,(\texttt{append})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau}{\Gamma \mid \Gamma', f_2 : \tau \vdash \texttt{move}\ f_1\ f_2 : \text{void}}\,(\texttt{move})$$

Figure 5.5: Typing rules

Example 5.7.3. The command `cp` f_1 f_2 is typable in the context $\Gamma = \{f_1 : \text{LC}^2, f_2 : \text{UC}\}$ since $f_1 \in \Gamma$ and $f_2 \in \Gamma$, and $T(f_1) \sqsubseteq \text{LC}^{n>0}$, as shown below.

$$\frac{\dfrac{}{f_1 : \text{LC}^2, f_2 : \text{UC} \mid f_2 : \text{UC} \vdash f_1 : \text{LC}^2}\,(f) \quad \text{LC}^2 \sqsubseteq \text{LC}^{n>0} \quad \dfrac{}{f_2 : \text{UC} \mid \varnothing \vdash f_2 : \text{UC}}\,(f)}{f_1 : \text{LC}^2, f_2 : \text{UC} \mid f_1 : \text{LC}^1, f_2 : \text{NC} \vdash \texttt{cp}\ f_1\ f_2 : \text{void}}\,(\texttt{cp})$$

Example 5.7.4. The command `cp` f_1 f_2 is not typable in the context $\Gamma = \{f_1 : \text{NC}, f_2 : \text{UC}\}$ since $T(f_1) \not\sqsubseteq \text{LC}^{n>0}$ as shown below.

$$\frac{\dfrac{}{f_1 : \text{NC}, f_2 : \text{UC} \mid f_2 : \text{UC} \vdash f_1 : \text{NC}}\,(f) \quad \text{NC} \not\sqsubseteq \text{LC}^{n>0} \quad \dfrac{}{f_2 : \text{UC} \mid \varnothing \vdash f_2 : \text{UC}}\,(f)}{f_1 : \text{NC}, f_2 : \text{UC} \mid ? \vdash \texttt{cp}\ f_1\ f_2 : ?}\,(?)$$

Example 5.7.5. The command `cp` f_1 f_2 is not typable in the context $\Gamma = \{f_1 : \text{UC}\}$ since $f_2 \notin \Gamma$ as shown below.

$$\dfrac{\dfrac{}{f_1 : \mathrm{UC} \mid \varnothing \vdash f_1 : \mathrm{UC}}\,(f) \qquad \mathrm{UC} \sqsubseteq \mathrm{LC}^{n>0} \qquad \dfrac{}{\varnothing \mid ? \vdash f_2 :?}\,(?)}{f_1 : \mathrm{UC} \mid ? \vdash \texttt{cp}\ f_1\ f_2 :?}\,(?)$$

5.7.3 Typing rule for `rm` command

$$\dfrac{\Gamma \mid \Gamma' \vdash f : \tau}{\Gamma \mid \Gamma' \vdash \texttt{rm}\ f : \mathrm{void}}\,(\texttt{rm})$$

The typing rule for `rm` command says that if we can type f from the context Γ, then we can type the command `rm` f of type void.

Example 5.7.6. The command `rm` f_1 is typable in the context $\Gamma = \{f_1 : \mathrm{NC}, f_2 : \mathrm{UC}\}$ since $f_1 \in \Gamma$ as shown below.

$$\dfrac{\dfrac{}{f_1 : \mathrm{NC}, f_2 : \mathrm{UC} \mid f_2 : \mathrm{UC} \vdash f_1 : \mathrm{NC}}\,(f)}{f_1 : \mathrm{NC}, f_2 : \mathrm{UC} \mid f_2 : \mathrm{UC} \vdash \texttt{rm}\ f_1 : \mathrm{void}}\,(\texttt{rm})$$

Example 5.7.7. The command `rm` f_1 is not typable in the context $\Gamma = \{f_2 : \mathrm{UC}\}$ since $f_1 \notin \Gamma$ as shown below.

$$\dfrac{\dfrac{}{f_2 : \mathrm{UC} \mid ? \vdash f_1 :?}\,(?)}{f_2 : \mathrm{UC} \mid ? \vdash \texttt{rm}\ f_1 :?}\,(?)$$

5.7.4 Typing rule for `mkf` command

$$\dfrac{}{\Gamma \mid \Gamma, f : t \vdash \texttt{mkf}\ f\ t : \mathrm{void}}\,(\texttt{mkf})$$

The typing rule for `mkf` command says that typing the command `mkf` f t of type void will add f of type t to the context Γ, provided that $f \notin \Gamma$.

Example 5.7.8. The command `mkf` f_1 NC is typable in the context $\Gamma = \{f_2 : \mathrm{UC}\}$ since $f_1 \notin \Gamma$ as shown below.

$$\dfrac{}{f_2 : \mathrm{UC} \mid f_2 : \mathrm{UC}, f_1 : \mathrm{NC} \vdash \texttt{mkf}\ f_1\ \mathrm{NC} : \mathrm{void}}\,(\texttt{mkf})$$

Example 5.7.9. The command `mkf` f_2 NC is not typable in the context $\Gamma = \{f_2 : \mathrm{UC}\}$ since $f_2 \in \Gamma$ as shown below.

$$\dfrac{}{f_2 : \mathrm{UC} \mid f_2 : \mathrm{UC}, ? \vdash \texttt{mkf}\ f_2\ \mathrm{NC} :?}\,(?)$$

5.7.5 Typing rule for `rd` command:

$$\frac{\Gamma \mid \Gamma' \vdash f : \tau}{\Gamma \mid \Gamma' \vdash \texttt{rd}\ f : \text{void}}\ (\texttt{rd})$$

The typing rule for `rd` command says that if we can type f from the context Γ, then we can type the command `rd` f of type void.

Example 5.7.10. The command `rd` f_1 is typable in the context $\Gamma = \{f_1 : \text{NC}, f_2 : \text{UC}\}$ since $f_1 \in \Gamma$ as shown below.

$$\frac{\dfrac{}{f_1 : \text{NC}, f_2 : \text{UC} \mid f_2 : \text{UC} \vdash f_1 : \text{NC}}\ (f)}{f_1 : \text{NC}, f_2 : \text{UC} \mid f_2 : \text{UC} \vdash \texttt{rd}\ f_1 : \text{void}}\ (\texttt{rd})$$

Example 5.7.11. The command `rd` f_1 is not typable in the context $\Gamma = \{f_2 : \text{UC}\}$ since $f_1 \notin \Gamma$ as shown below.

$$\frac{\dfrac{}{f_2 : \text{UC} \mid ? \vdash f_1 : ?}\ (?)}{f_2 : \text{UC} \mid ? \vdash \texttt{rd}\ f_1 : ?}\ (?)$$

5.7.6 Typing rule for `cat` command

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau' \quad \Gamma'' \mid \Gamma''' \vdash f_3 : \tau''}{\Gamma \mid \Gamma''', f_3 : \tau \sqcup \tau' \sqcup \tau'' \vdash \texttt{cat}\ f_1\ f_2\ f_3 : \text{void}}\ (\texttt{cat})$$

The typing rule for `cat` command says that if we can type f_1, f_2 and f_3 from the context Γ, then we can type the command `cat` f_1 f_2 f_3 of type void and f_1 and f_2 will be consumed from the context Γ while the type of f_3 is changed to be the least upper bound of its type, the type of f_1 and the type of f_2.

Example 5.7.12. The command `cat` f_1 f_2 f_3 is typable in the context $\Gamma = \{f_1 : \text{UC}, f_2 : \text{NC}, f_3 : \text{LC}^4\}$ since $f_1 \in \Gamma$ and $f_2 \in \Gamma$ and $f_3 \in \Gamma$ as shown below. To compress the proof, let $\Delta = \{f_2 : \text{NC}, f_3 : \text{LC}^4\}$.

$$\frac{\dfrac{}{\Gamma \mid \Delta \vdash f_1 : \text{UC}}\ (f) \quad \dfrac{}{\Delta \mid f_3 : \text{LC}^4 \vdash f_2 : \text{NC}}\ (f) \quad \dfrac{}{f_3 : \text{LC}^4 \mid \varnothing \vdash f_3 : \text{LC}^4}\ (f)}{\Gamma \mid f_3 : \text{NC} \vdash \texttt{cat}\ f_1\ f_2\ f_3 : \text{void}}\ (\texttt{cat})$$

Example 5.7.13. The command `cat` f_1 f_2 f_3 is not typable in the context $\Gamma = \{f_1 : \text{UC}, f_2 : \text{NC}\}$ since $f_3 \notin \Gamma$ as shown below.

$$\frac{\dfrac{}{f_1 : \text{UC}, f_2 : \text{NC} \mid f_2 : \text{NC} \vdash f_1 : \text{UC}}\ (f) \quad \dfrac{}{f_2 : \text{NC} \mid \varnothing \vdash f_2 : \text{NC}}\ (f) \quad \dfrac{}{\varnothing \mid ? \vdash f_3 : ?}\ (?)}{f_1 : \text{UC}, f_2 : \text{NC} \mid ? \vdash \texttt{cat}\ f_1\ f_2\ f_3 : ?}\ (?)$$

5.7.7 Typing rule for `mv` command

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau'}{\Gamma \mid \Gamma'', f_2 : \tau \sqcup \tau' \vdash \texttt{mv}\ f_1\ f_2 : \text{void}}\ (\texttt{mv})$$

The typing rule for `mv` command says that if we can type f_1 and f_2 from the context Γ, then we can type the command `mv` f_1 f_2 of type void and f_1 will be consumed from the context Γ while the type of f_2 is changed to be the least upper bound of its type and the type of f_1.

Example 5.7.14. The command `mv` f_1 f_2 is typable in the context $\Gamma = \{f_1 : \text{NC}, f_2 : \text{UC}\}$ since $f_1 \in \Gamma$ and $f_2 \in \Gamma$ as shown below.

$$\frac{\dfrac{}{f_1 : \text{NC}, f_2 : \text{UC} \mid f_2 : \text{UC} \vdash f_1 : \text{NC}}\ (f) \quad \dfrac{}{f_2 : \text{UC} \mid \varnothing \vdash f_2 : \text{UC}}\ (f)}{f_1 : \text{NC}, f_2 : \text{UC} \mid f_2 : \text{NC} \vdash \texttt{mv}\ f_1\ f_2 : \text{void}}\ (\texttt{mv})$$

Example 5.7.15. The command `mv` f_1 f_2 is not typable in the context $\Gamma = \{f_1 : \text{NC}\}$ since $f_2 \notin \Gamma$ as shown below.

$$\frac{\dfrac{}{f_1 : \text{NC} \mid \varnothing \vdash f_1 : \text{NC}}\ (f) \quad \dfrac{}{\varnothing \mid ? \vdash f_2 : ?}\ (?)}{f_1 : \text{NC} \mid ? \vdash \texttt{mv}\ f_1\ f_2 : ?}\ (?)$$

5.7.8 Typing rule for `copy` command

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \tau \sqsubseteq \text{LC}^{n>0}}{\Gamma \mid \Gamma', f_1 : \mathit{red}(\tau), f_2 : \mathit{dst}(\tau) \vdash \texttt{copy}\ f_1\ f_2 : \text{void}}\ (\texttt{copy})$$

The typing rule for `copy` command says that if we can type f_1 from the context Γ, and f_1 is of type UC or $\text{LC}^{n>0}$, then we can type the command `copy` f_1 f_2 of type void and the type of f_1 is changed to be $\mathit{red}(T(f_1))$, and f_2 will added to the context Γ and assigned the type $\mathit{dst}(f_1)$, provided that $f_2 \notin \Gamma$.

Example 5.7.16. The command `copy` f_1 f_2 is typable in the context $\Gamma = \{f_1 : \text{LC}^1\}$ since $f_1 \in \Gamma$, $f_2 \notin \Gamma$ and $T(f_1) \sqsubseteq \text{LC}^{n>0}$ as shown below.

$$\frac{\dfrac{}{f_1 : \text{LC}^1 \mid \varnothing \vdash f_1 : \text{LC}^1}\ (f) \quad \text{LC}^1 \sqsubseteq \text{LC}^{n>0}}{f_1 : \text{LC}^1 \mid\ f_1 : \text{LC}^0, f_2 : \text{NC} \vdash \texttt{copy}\ f_1\ f_2 : \text{void}}\ (\texttt{copy})$$

Example 5.7.17. The command `copy` f_1 f_2 is not typable in the context $\Gamma = \{f_1 : \text{LC}^0\}$ since $T(f_1) \not\sqsubseteq \text{LC}^{n>0}$ as shown below.

$$\frac{\dfrac{}{f_1 : \text{LC}^0 \mid \varnothing \vdash f_1 : \text{LC}^0}\ (f) \quad \text{LC}^0 \not\sqsubseteq \text{LC}^{n>0}}{f_1 : \text{LC}^0 \mid ? \vdash \texttt{copy}\ f_1\ f_2 : ?}\ (?)$$

Example 5.7.18. The command `copy` f_1 f_2 is not typable in the context $\Gamma = \{f_1 : \mathrm{LC}^1, f_2 : \mathrm{UC}\}$ since $f_2 \in \Gamma$ as shown below.

$$\dfrac{\dfrac{}{f_1 : \mathrm{LC}^1, f_2 : \mathrm{UC} \mid f_2 : \mathrm{UC} \vdash f_1 : \mathrm{LC}^1}\,(f) \qquad \mathrm{LC}^1 \sqsubseteq \mathrm{LC}^{n>0}}{f_1 : \mathrm{LC}^1 \mid f_2 : \mathrm{UC},\ f_1 : \mathrm{LC}^1, ? \vdash \texttt{copy}\ f_1\ f_2 :?}\,(?)$$

Example 5.7.19. The command `copy` f_1 f_2 is not typable in the context $\Gamma = \varnothing$ since $f_1 \notin \Gamma$ as shown below.

$$\dfrac{\dfrac{}{\varnothing \mid ? \vdash f_1 :?}\,(?) \qquad ? \sqsubseteq \mathrm{LC}^{n>0}}{\varnothing \mid ? \vdash \texttt{copy}\ f_1\ f_2 :?}\,(?)$$

5.7.9 Typing rule for `append` command

$$\dfrac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau'}{\Gamma \mid \Gamma'', f_3 : \tau \sqcup \tau' \vdash \texttt{append}\ f_1\ f_2\ f_3 : \mathrm{void}}\,(\texttt{append})$$

The typing rule for `append` command says that if we can type f_1 and f_2 from the context Γ, then we can type the command `append` f_1 f_2 f_3 of type void and f_1 and f_2 will be consumed from the context Γ while f_3 will be added to the context Γ and its type will be the least upper bound of the type of f_1 and the type of f_2, provided that $f_3 \notin \Gamma$.

Example 5.7.20. The command `append` f_1 f_2 f_3 is typable in the context $\Gamma = \{f_1 : \mathrm{UC}, f_2 : \mathrm{NC}\}$ since $f_1 \in \Gamma$, $f_2 \in \Gamma$, and $f_3 \notin \Gamma$ as shown below.

$$\dfrac{\dfrac{}{f_1 : \mathrm{UC}, f_2 : \mathrm{NC} \mid f_2 : \mathrm{NC} \vdash f_1 : \mathrm{UC}}\,(f) \qquad \dfrac{}{f_2 : \mathrm{NC} \mid \varnothing \vdash f_2 : \mathrm{NC}}\,(f)}{f_1 : \mathrm{UC}, f_2 : \mathrm{NC} \mid f_3 : \mathrm{NC} \vdash \texttt{append}\ f_1\ f_2\ f_3 : \mathrm{void}}\,(\texttt{append})$$

Example 5.7.21. The command `append` f_1 f_2 f_3 is not typable in the context $\Gamma = \{f_1 : \mathrm{UC}, f_2 : \mathrm{NC}, f_3 : \mathrm{LC}^0\}$ since $f_3 \in \Gamma$ as shown below.

$$\dfrac{\dfrac{}{\Gamma \mid f_2 : \mathrm{NC}, f_3 : \mathrm{LC}^0 \vdash f_1 : \mathrm{UC}}\,(f) \qquad \dfrac{}{f_2 : \mathrm{NC}, f_3 : \mathrm{LC}^0 \mid f_3 : \mathrm{LC}^0 \vdash f_2 : \mathrm{NC}}\,(f)}{\Gamma \mid f_3 : \mathrm{LC}^0, ? \vdash \texttt{append}\ f_1\ f_2\ f_3 : \mathrm{void}}\,(?)$$

Example 5.7.22. The command `append` f_1 f_2 f_3 is not typable in the context $\Gamma = \{f_1 : \mathrm{UC}\}$ since $f_2 \notin \Gamma$ as shown below.

$$\dfrac{\dfrac{}{f_1 : \mathrm{UC} \mid \varnothing \vdash f_1 : \mathrm{UC}}\,(f) \qquad \dfrac{}{\varnothing \mid ? \vdash f_2 :?}\,(?)}{f_1 : \mathrm{UC} \mid ? \vdash \texttt{append}\ f_1\ f_2\ f_3 :?}\,(?)$$

5.7.10 Typing rule for `move` command

$$\dfrac{\Gamma \mid \Gamma' \vdash f_1 : \tau}{\Gamma \mid \Gamma', f_2 : \tau \vdash \texttt{move}\ f_1\ f_2 : \text{void}}\ (\texttt{move})$$

The typing rule for `move` command says that if we can type f_1 from the context Γ, then we can type the command `move` f_1 f_2 of type void and f_1 will be consumed from the context Γ while f_2 will be added to the context Γ and its type will be the same as the type of f_1, provided that $f_2 \notin \Gamma$.

Example 5.7.23. The command `move` f_1 f_2 is typable in the context $\Gamma = \{f_1 : \text{NC}\}$ since $f_1 \in \Gamma$ and $f_2 \notin \Gamma$ as shown below.

$$\dfrac{\dfrac{}{f_1 : \text{NC} \mid \varnothing \vdash f_1 : \text{NC}}\ (f)}{f_1 : \text{NC} \mid f_2 : \text{NC} \vdash \texttt{move}\ f_1\ f_2 : \text{void}}\ (\texttt{move})$$

Example 5.7.24. The command `move` f_1 f_2 is not typable in the context $\Gamma = \{f_1 : \text{NC}, f_2 : \text{UC}\}$ since $f_2 \in \Gamma$ as shown below.

$$\dfrac{\dfrac{}{f_1 : \text{NC}, f_2 : \text{UC} \mid f_2 : \text{UC} \vdash f_1 : \text{NC}}\ (f)}{f_1 : \text{NC}, f_2 : \text{UC} \mid f_2 : \text{UC}, ? \vdash \texttt{move}\ f_1\ f_2 :?}\ (?)$$

Example 5.7.25. The command `move` f_1 f_2 is not typable in the context $\Gamma = \varnothing$ since $f_1 \notin \Gamma$ as shown below.

$$\dfrac{\dfrac{}{\varnothing \mid ? \vdash f_1 :?}\ (?)}{\varnothing \mid ? \vdash \texttt{move}\ f_1\ f_2 :?}\ (?)$$

5.7.11 Typing rule for sequences of commands

$$\dfrac{\Gamma \mid \Gamma' \vdash c : \text{void} \quad \Gamma' \mid \Gamma'' \vdash cs : \text{void}}{\Gamma \mid \Gamma'' \vdash c; cs : \text{void}}\ (cs)$$

The typing rule for sequences of commands cs says that if typing the command c of type void changes the context Γ to Γ' and typing the command cs of type void changes the context Γ' to Γ'', then typing these commands in sequence changes the context Γ to Γ''.

Example 5.7.26. The sequence of commands `rd` f_1;`rd`f_2;`rd` f_3 is typable in the context $\Gamma = \{f_1 : \text{UC}, f_2 : \text{LC}^2, f_3 : \text{NC}\}$ as shown below. To compress the proof, let $\Delta = \{f_2 : \text{LC}^2, f_3 : \text{NC}\}$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \mid \Delta \vdash f_1 : \text{UC}}\ (f)}{\Gamma \mid \Delta \vdash \texttt{rd}\ f_1 : \text{void}}\ (\texttt{rd}) \quad \dfrac{\dfrac{\dfrac{}{\Delta \mid f_3 : \text{NC} \vdash f_2 : \text{LC}^2}\ (f)}{\Delta \mid f_3 : \text{NC} \vdash \texttt{rd}\ f_2 : \text{void}}\ (\texttt{rd}) \quad \dfrac{\dfrac{}{f_3 : \text{NC} \mid \varnothing \vdash f_3 : \text{NC}}\ (f)}{f_3 : \text{NC} \mid \varnothing \vdash \texttt{rd}\ f_3 : \text{void}}\ (\texttt{rd})}{\Delta \mid \varnothing \vdash \texttt{rd}f_2; \texttt{rd}\ f_3 : \text{void}}\ (\texttt{rd})}{f_1 : \text{UC}, f_2 : \text{LC}^2, f_3 : \text{NC} \mid \varnothing \vdash \texttt{rd}\ f_1; \texttt{rd}f_2; \texttt{rd}\ f_3 : \text{void}}\ (cs)$$

Example 5.7.28. The commands $\texttt{mkf}\ f_1\ \mathrm{LC}^2; \texttt{copy}\ f_1\ f_2; \texttt{append}\ f_1\ f_2\ f_3; \texttt{move} f_3\ f_4; \texttt{rd}\ f_4$ are typable in the context $\Gamma = \varnothing$. To compress the proof let, $\Delta = \{f_1 : \mathrm{LC}^1, f_2 : \mathrm{NC}\}$ and $\Delta' = \{f_3 : \mathrm{NC}\}$.

$$\dfrac{\dfrac{}{\varnothing \mid f_1 : \mathrm{LC}^2 \vdash \texttt{mkf}\ f_1\ \mathrm{LC}^2 : \mathrm{void}}\,(\texttt{mkf}) \quad \dfrac{\dfrac{\dfrac{}{f_1 : \mathrm{LC}^2 \mid \varnothing \vdash f_1 : \mathrm{LC}^2}\,(f) \quad \mathrm{LC}^2 \sqsubseteq \mathrm{LC}^{N>0}}{f_1 : \mathrm{LC}^2 \mid \Delta \vdash \texttt{copy}\ f_1\ f_2 : \mathrm{void}}\,(\texttt{copy}) \quad \dfrac{\dfrac{\dfrac{}{\Delta \mid f_2 : \mathrm{NC} \vdash f_1 : \mathrm{LC}^1}\,(f) \quad \dfrac{}{f_2 : \mathrm{NC} \mid \varnothing \vdash f_2 : \mathrm{NC}}\,(f)}{\Delta \mid \Delta' \vdash \texttt{append}\ f_1\ f_2\ f_3 : \mathrm{void}}\,(\texttt{append}) \quad \dfrac{\dfrac{\dfrac{}{\Delta' \mid \varnothing \vdash f_3 : \mathrm{NC}}\,(f)}{\Delta' \mid f_4 : \mathrm{NC} \vdash \texttt{move} f_3\ f_4 : \mathrm{void}}\,(\texttt{move}) \quad \dfrac{\dfrac{}{f_4 : \mathrm{NC} \mid \varnothing \vdash\ f_4 : \mathrm{NC}}\,(f)}{f_4 : \mathrm{NC} \mid \varnothing \vdash \texttt{rd}\ f_4 : \mathrm{void}}\,(\texttt{rd})}{\Delta' \mid \varnothing \vdash \texttt{move} f_3\ f_4; \texttt{rd}\ f_4 : \mathrm{void}}\,(cs)}{\Delta \mid \varnothing \vdash \texttt{append}\ f_1\ f_2\ f_3; \texttt{move} f_3\ f_4; \texttt{rd}\ f_4 : \mathrm{void}}\,(cs)}{f_1 : \mathrm{LC}^2 \mid \varnothing \vdash \texttt{copy}\ f_1\ f_2; \texttt{append}\ f_1\ f_2\ f_3; \texttt{move} f_3\ f_4; \texttt{rd}\ f_4 : \mathrm{void}}\,(cs)}{\varnothing \mid \varnothing \vdash \texttt{mkf}\ f_1\ \mathrm{LC}^2; \texttt{copy}\ f_1\ f_2; \texttt{append}\ f_1\ f_2\ f_3; \texttt{move} f_3\ f_4; \texttt{rd}\ f_4 : \mathrm{void}}\,(cs)$$

Example 5.7.27. The sequence of commands $\texttt{copy}\ f_1\ f_2; \texttt{rd}\ f_2; \texttt{rm}\ f_1$ is typable in the context $\Gamma = \{f_1 : \mathrm{LC}^2\}$ as shown below. To compress the proof, let $\Delta = \{f_1 : \mathrm{LC}^1, f_2 : \mathrm{NC}\}$, $\Delta' = \{f_1 : \mathrm{LC}^1\}$ and $C = \mathrm{LC}^2 \sqsubseteq \mathrm{LC}^{n>0}$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \mid \varnothing \vdash f_1 : \mathrm{LC}^2}\,(f) \quad C}{\Gamma \mid \Delta \vdash \texttt{copy}\ f_1\ f_2 : \mathrm{void}}\,(\texttt{copy}) \quad \dfrac{\dfrac{\dfrac{}{\Delta \mid \Delta' \vdash f_2 : \mathrm{NC}}\,(f)}{\Delta \mid \Delta' \vdash \texttt{rd}\ f_2 : \mathrm{void}}\,(\texttt{rd}) \quad \dfrac{\dfrac{}{\Delta' \mid \varnothing \vdash f_1 : \mathrm{LC}^1}\,(f)}{\Delta' \mid \varnothing \vdash \texttt{rm}\ f_1 : \mathrm{void}}\,(\texttt{rm})}{\Delta \mid \varnothing \vdash \texttt{rd}\ f_2; \texttt{rm}\ f_1 : \mathrm{void}}\,(cs)}{\Gamma \mid \varnothing \vdash \texttt{copy}\ f_1\ f_2; \texttt{rd}\ f_2; \texttt{rm}\ f_1 : \mathrm{void}}\,(cs)$$

5.8 Properties of the type system

In this section we prove the soundness of our type system with respect to the operational semantics. A type system is sound if well-typed programs compute without evaluation errors. We defined evaluation errors to be syntactical and type errors. Syntactical errors can occur by all commands while type errors can only occur by two commands which are `cp` and `copy`. This is because `cp` and `copy` commands apply the operations $\mathit{dst}(\tau)$ and $\mathit{red}(\tau)$ to the type of the source file and will cause a type error if their constraints are not satisfied. Therefore, all well-typed commands evaluate without an error if they can evaluate without syntactical errors and well-typed `cp` and `copy` commands evaluate without an error if they can evaluate without syntactical errors as well as type errors. We prove the soundness of our type system by proving two properties which are progress and preservation.

5.8.1 Progress

Traditionally, the progress theorem states that a program is either a value or can take a step of evaluation. However, in our case, programs are commands that operate on files in a file system, and should always take a step of evaluation. Therefore, if a command e is typable in a particular file system δ, then the command e must take a step of evaluation.

Theorem 5.8.1 (Progress)**.** If $\Gamma = \delta$ and $\Gamma \mid \Gamma' \vdash e : \tau$, then $\langle e, \delta\rangle \not\to \mathit{Err}$

Proof. We proceed by cases on typing derivation of e. There are 9 cases as follows.

1. $e =$ `cp` $f_1\ f_2$

   We know there is a typing derivation for e by using rule (`cp`) with conclusion: $\Gamma \mid \Gamma'', f_1 : \mathit{red}(\tau), f_2 : \tau' \sqcup \mathit{dst}(\tau) \vdash$ `cp` $f_1\ f_2$: void. We must also have subderivations with conclusions: $\Gamma \mid \Gamma' \vdash f_1 : \tau$, $\tau \sqsubseteq \mathrm{LC}^{n>0}$ and $\Gamma' \mid \Gamma'' \vdash f_2 : \tau'$. Now we can use rule (1) to obtain $\langle$`cp` $f_1\ f_2\ , \delta\rangle \to \delta[f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_2) \sqcup \mathit{dst}(T(f_1))][f_1 \leftarrow \mathit{red}(T(f_1))]$. Since the configuration $\langle$`cp` $f_1\ f_2\ , \delta\rangle$ require $f_1 \in \delta$ and $f_2 \in \delta$ and $f_1 \neq f_2$ to be evaluated without syntactical error, and we have $f_1 \in \Gamma$ and $f_2 \in \Gamma$ and $f_1 \neq f_2$ in Γ, because Γ does not allow repetition of file names, and $\Gamma = \delta$. Then, $\langle$`cp` $f_1\ f_2\ , \delta\ \rangle \not\to^s \mathit{Err}$. Also, since the operations $\mathit{dst}(T(f_1))$ and $\mathit{red}(T(f_1))$ requires $(T(f_1)) \sqsubseteq \mathrm{LC}^{n>0}$ in δ and we have $T(f_1) \sqsubseteq \mathrm{LC}^{n>0}$ in Γ and $\Gamma = \delta$. Then, $\langle$`cp` $f_1\ f_2\ , \delta\ \rangle \not\to \mathit{Err}$ as required.

2. $e =$ `rm` f

We know there is a typing derivation for e by using rule (`rm`) with conclusion: $\Gamma \mid \Gamma' \vdash \texttt{rm}\ f : \text{void}$. We must also have subderivation with conclusion: $\Gamma \mid \Gamma' \vdash f : \tau$. Now we can use rule (2) to obtain $\langle \texttt{rm}\ f, \delta\rangle \rightarrow \delta[-f]$. Since the configuration $\langle \texttt{rm}\ f, \delta\rangle$ requires $f \in \delta$ to be evaluated without syntactical error, and we have $f \in \Gamma$ and $\Gamma = \delta$. Then $\langle \texttt{rm}\ f, \delta\rangle \not\rightarrow \mathit{Err}$.

3. $e = \texttt{mkf}\ f\ t$

   We know there is a typing derivation for e by using rule (`mkf`) with conclusion: $\Gamma \mid \Gamma, f : t \vdash \texttt{mkf}\ f\ t : \text{void}$. Now we can use rule (3) to obtain $\langle \texttt{mkf}\ f\ t, \delta\rangle \rightarrow \delta[+f][f \leftarrow t]$. Since the configuration $\langle \texttt{mkf}\ f\ t, \delta\rangle$ requires $f \notin \delta$ to be evaluated without syntactical error, and we have $f \notin \Gamma$, because the symbol "," in $\Gamma, f : t$ does not allow repetition of file names, and $\Gamma = \delta$. Then, $\langle \texttt{mkf}\ f\ t, \delta\rangle \not\rightarrow \mathit{Err}$.

4. $e = \texttt{rd}\ f$

   We know there is a typing derivation for e by using rule (`rd`) with conclusion: $\Gamma \mid \Gamma' \vdash \texttt{rd}\ f : \text{void}$. We must also have subderivation with conclusion: $\Gamma \mid \Gamma' \vdash f : \tau$. Now we can use rule (4) to obtain $\langle \texttt{rd}\ f, \delta\rangle \rightarrow \delta[-f]$. Since the configuration $\langle \texttt{rd}\ f, \delta\rangle$ requires $f \in \delta$ to be evaluated without syntactical error, and we have $f \in \Gamma$ and $\Gamma = \delta$. Then $\langle \texttt{rd}\ f, \delta\rangle \not\rightarrow \mathit{Err}$.

5. $e = \texttt{cat}\ f_1\ f_2\ f_3$

   We know there is a typing derivation for e by using rule (`cat`) with conclusion: $\Gamma \mid \Gamma''', f_3 : \tau \sqcup \tau' \sqcup \tau'' \vdash \texttt{cat}\ f_1\ f_2\ f_3 : \text{void}$. We must also have subderivations with conclusions: $\Gamma \mid \Gamma' \vdash f_1 : \tau$, $\Gamma' \mid \Gamma'' \vdash f_2 : \tau'$, and $\Gamma'' \mid \Gamma''' \vdash f_3 : \tau''$. Now we can use rule (5) to obtain $\langle \texttt{cat}\ f_1\ f_2\ f_3, \delta\rangle \rightarrow \delta[f_3 \leftarrow C(f_1) + C(f_2)][f_3 \leftarrow T(f_1) \sqcup T(f_2) \sqcup T(f_3)][-f_1, -f_2]$. Since the configuration $\langle \texttt{cat}\ f_1\ f_2\ f_3, \delta\rangle$ requires $f_1 \in \delta$, $f_2 \in \delta$, $f_3 \in \delta$, and f_1, f_2 and f_3 have distinct names to be evaluated without syntactical error, and we have $f_1 \in \Gamma$, $f_2 \in \Gamma$, $f_3 \in \Gamma$, and f_1, f_2 and f_3 have distinct names in Γ, because Γ does not allow repetition of file names, and $\Gamma = \delta$. Then, $\langle \texttt{cat}\ f_1\ f_2\ f_3, \delta\rangle \not\rightarrow \mathit{Err}$.

6. $e = \texttt{mv}\ f_1\ f_2$

   We know there is a typing derivation for e by using rule (`mv`) with conclusion: $\Gamma \mid \Gamma'', f_2 : \tau \sqcup \tau' \vdash \texttt{mv}\ f_1\ f_2 : \text{void}$. We must also have subderivations with conclusions: $\Gamma \mid \Gamma' \vdash f_1 : \tau$ and $\Gamma' \mid \Gamma'' \vdash f_2 : \tau'$. Now we can use rule (6) to obtain $\langle \texttt{mv}\ f_1\ f_2\ , \delta\rangle \rightarrow \delta[f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_1) \sqcup T(f_2)][-f_1]$. Since the configuration $\langle \texttt{mv}\ f_1\ f_2\ , \delta\rangle$ requires

$f_1 \in \delta$, $f_2 \in \delta$, and $f_1 \neq f_2$ in δ to be evaluated without syntactical error, and we have $f_1 \in \Gamma$, $f_2 \in \Gamma$, and $f_1 \neq f_2$ in Γ, because Γ does not allow repetition of file names, and $\Gamma = \delta$. Then, $\langle \texttt{mv}\ f_1\ f_2, \delta\rangle \not\rightarrow \mathit{Err}$.

7. $e = \texttt{copy}\ f_1\ f_2$

   We know there is a typing derivation for e by using rule ($\texttt{copy}$) with conclusion: $\Gamma \mid \Gamma', f_1 : \mathit{red}(\tau), f_2 : \mathit{dst}(\tau) \vdash \texttt{copy}\ f_1\ f_2 : \text{void}$. We must also have subderivation with conclusion: $\Gamma \mid \Gamma' \vdash f_1 : \tau$ and $\tau \sqsubseteq \text{LC}^{n>0}$. Now we can use rule (7) to obtain $\langle \texttt{copy}\ f_1\ f_2\ , \delta\rangle \rightarrow \delta[+f_2, f_2 \leftarrow C(f_1)][f_2 \leftarrow \mathit{dst}(T(f_1))][f_1 \leftarrow \mathit{red}(T(f_1))]$. Since the configuration $\langle \texttt{copy}\ f_1\ f_2\ , \delta\rangle$ requires $f_1 \in \delta$, $f_2 \notin \delta$ and $f_1 \neq f_2$ in δ to be evaluated without syntactical error, and we have $f_1 \in \Gamma$, $f_2 \notin \Gamma$ and $f_1 \neq f_2$ in Γ, and $\Gamma = \delta$. Then, $\langle \texttt{copy}\ f_1\ f_2\ , \delta\rangle \not\rightarrow^s \mathit{Err}$. Also, since the operations $\mathit{dst}(T(f_1))$ and $\mathit{red}(T(f_1))$ requires $(T(f_1)) \sqsubseteq \text{LC}^{n>0}$ in δ and we have $T(f_1) \sqsubseteq \text{LC}^{n>0}$ in Γ and $\Gamma = \delta$. Then, $\langle \texttt{copy}\ f_1\ f_2\ , \delta\ \rangle \not\rightarrow \mathit{Err}$ as required.

8. $e = \texttt{append}\ f_1\ f_2\ f_3$

   We know there is a typing derivation for e by using rule ($\texttt{append}$) with conclusion: $\Gamma \mid \Gamma'', f_3 : \tau \sqcup \tau' \vdash \texttt{append}\ f_1\ f_2\ f_3 : \text{void}$. We must also have subderivations with conclusions: $\Gamma \mid \Gamma' \vdash f_1 : \tau$ and $\Gamma' \mid \Gamma'' \vdash f_2 : \tau'$. Now we can use rule (8) to obtain $\langle \texttt{append}\ f_1\ f_2\ f_3, \delta\rangle \rightarrow \delta[+f_3, f_3 \leftarrow C(f_1) + C(f_2)][f_3 \leftarrow T(f_1) \sqcup T(f_2)][-f_1, -f_2]$. Since the configuration $\langle \texttt{append}\ f_1\ f_2\ f_3, \delta\rangle$ requires $f_1 \in \delta$, $f_2 \in \delta$, $f_3 \notin \delta$, and f_1, f_2 and f_3 have distinct names in δ to be evaluated without syntactical error, and we have $f_1 \in \Gamma$, $f_2 \in \Gamma$, $f_3 \notin \Gamma$, and f_1, f_2 and f_3 have distinct names in Γ, because Γ does not allow repetition of file names, and $\Gamma = \delta$. Then, $\langle \texttt{append}\ f_1\ f_2\ f_3, \delta\rangle \not\rightarrow \mathit{Err}$.

9. $e = \texttt{move}\ f_1\ f_2$

   We know there is a typing derivation for e by using rule ($\texttt{move}$) with conclusion: $\Gamma \mid \Gamma', f_2 : \tau \vdash \texttt{move}\ f_1\ f_2 : \text{void}$. We must also have subderivation with conclusion: $\Gamma \mid \Gamma' \vdash f_1 : \tau$. Now we can use rule (9) to obtain $\langle \texttt{move}\ f_1\ f_2, \delta\ \rangle \rightarrow \delta[+f_2, f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_1)][-f_1]$. Since the configuration $\langle \texttt{move}\ f_1\ f_2, \delta\ \rangle$ requires $f_2 \notin \delta$, $f_1 \in \delta$, and $f_1 \neq f_2$ in δ, and we have $f_2 \notin \Gamma$, $f_1 \in \Gamma$, and $f_1 \neq f_2$ in Γ, and $\Gamma = \delta$. Then, $\langle \texttt{move}\ f_1\ f_2, \delta\rangle \not\rightarrow \mathit{Err}$.

$\square$

5.8.2 Preservation

Traditionally, the preservation theorem states that as we evaluated a program, its type is preserved at each evaluation step. As we mentioned above, in our case, programs are commands which are all of type void. However, programs manipulate files and their types, and we need to ensure that types of files are preserved during evaluation. Therefore, if a command is typable in a particular file system δ, then types of files we obtain by typing the command must be preserved in the file system we obtain by evaluating the command. This property shows the consistency of the type system with the operational semantics, that is not only typed commands evaluate without errors, but also the types of files in the file system after evaluating the command correspond to the types of files resulted from typing the commands.

Theorem 5.8.2 (Preservation)**.** If $\Gamma = \delta$ and $\Gamma \mid \Gamma' \vdash e : \tau$, and $\langle e, \delta\rangle \to \delta'$, then $\Gamma' = \delta'$.

Proof. We proceed by cases on $\langle e, \delta\rangle \to \delta'$. There are 9 cases as follows.

1. $e = \langle \texttt{cp}\ f_1\ f_2\ , \delta\ \rangle \to \delta[f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_2) \sqcup \mathit{dst}(T(f_1))][f_1 \leftarrow \mathit{red}(T(f_1))]$

   We know there is a typing derivation for e by using rule ($\texttt{cp}$) with conclusion: $\Gamma \mid \Gamma'', f_1 : \mathit{red}(\tau), f_2 : \tau' \sqcup \mathit{dst}(\tau) \vdash \texttt{cp}\ f_1\ f_2 : \text{void}$. We must also have subderivations with conclusions: $\Gamma \mid \Gamma' \vdash f_1 : \tau$, $\tau \sqsubseteq \text{LC}^{n>0}$ and $\Gamma' \mid \Gamma'' \vdash f_2 : \tau'$. To compress the proof let $\delta' = \delta[f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_2) \sqcup \mathit{dst}(T(f_1))][f_1 \leftarrow \mathit{red}(T(f_1))]$. Now we have the following cases based on typing f_1 and f_2.

   (a) : $\Gamma \mid \Gamma' \vdash f_1 : \text{LC}^{n>0}\quad \Gamma' \mid \Gamma'' \vdash f_2 : \text{UC}$

   In this case, the typing derivation of e must have the form $\Gamma \mid \Gamma'', f_1 : \text{LC}^{n-1},\ f_2 : \text{NC} \vdash \texttt{cp}\ f_1\ f_2 : \text{void}$. Let $\Gamma' = \Gamma'', f_1 : \text{LC}^{n-1},\ f_2 : \text{NC}$. Now we know that $\Gamma \neq \Gamma'$ because $\Gamma(f_1) \neq \Gamma'(f_1)$ and $\Gamma(f_2) \neq \Gamma'(f_2)$, that is $\text{LC}^{n>0} \neq \text{LC}^{n-1}$ and $\text{UC} \neq \text{NC}$, respectively. We also know that $\delta \neq \delta'$ because $\delta(f_1) \neq \delta'(f_1)$ and $\delta(f_2) \neq \delta'(f_2)$, that is $\text{LC}^{n>0} \neq \text{LC}^{n-1}$ and $\text{UC} \neq \text{NC}$, respectively. Since $\Gamma \neq \Gamma'$ because $\Gamma(f_1) \neq \Gamma'(f_1)$ and $\Gamma(f_2) \neq \Gamma'(f_2)$, and $\delta \neq \delta'$ because $\delta(f_1) \neq \delta'(f_1)$ and $\delta(f_2) \neq \delta'(f_2)$, and $\Gamma'(f_1) = \delta'(f_1)$ that is $\text{LC}^{n-1} = \text{LC}^{n-1}$ and $\Gamma'(f_2) = \delta'(f_2)$ that is $\text{NC} = \text{NC}$, and $\Gamma = \delta$. Then, $\Gamma' = \delta'$ as required.

   (b) : $\Gamma \mid \Gamma' \vdash f_1 : \text{LC}^{n>0}\quad \Gamma' \mid \Gamma'' \vdash f_2 : \text{LC}^{n>0}$.

   In this case, the typing derivation of e must have the form $\Gamma \mid \Gamma'', f_1 : \text{LC}^{n-1},\ f_2 : \text{NC} \vdash \texttt{cp}\ f_1\ f_2 : \text{void}$. Let $\Gamma' = \Gamma'', f_1 : \text{LC}^{n-1},\ f_2 : \text{NC}$. Now we know that $\Gamma \neq \Gamma'$ because $\Gamma(f_1) \neq \Gamma'(f_1)$ and $\Gamma(f_2) \neq \Gamma'(f_2)$, that is $\text{LC}^{n>0} \neq \text{LC}^{n-1}$ and

$\text{LC}^{n>0} \neq \text{NC}$, respectively. We also know that $\delta \neq \delta'$ because $\delta(f_1) \neq \delta'(f_1)$ and $\delta(f_2) \neq \delta'(f_2)$, that is $\text{LC}^{n>0} \neq \text{LC}^{n-1}$ and $\text{LC}^{n>0} \neq \text{NC}$, respectively. Since $\Gamma \neq \Gamma'$ because $\Gamma(f_1) \neq \Gamma'(f_1)$ and $\Gamma(f_2) \neq \Gamma'(f_2)$, and $\delta \neq \delta'$ because $\delta(f_1) \neq \delta'(f_1)$ and $\delta(f_2) \neq \delta'(f_2)$, and $\Gamma'(f_1) = \delta'(f_1)$ that is $\text{LC}^{n-1} = \text{LC}^{n-1}$ and $\Gamma'(f_2) = \delta'(f_2)$ that is $\text{NC} = \text{NC}$, and $\Gamma = \delta$. Then, $\Gamma' = \delta'$ as required.

(c) $: \Gamma \mid \Gamma' \vdash f_1 : \text{LC}^{n>0} \quad \Gamma' \mid \Gamma'' \vdash f_2 : \text{NC}$

In this case, the typing derivation of e must have the form $\Gamma \mid \Gamma'', f_1 : \text{LC}^{n-1}, f_2 : \text{NC} \vdash \texttt{cp}\ f_1\ f_2 : \text{void}$. Let $\Gamma' = \Gamma'', f_1 : \text{LC}^{n-1}, f_2 : \text{NC}$. Now we know that $\Gamma \neq \Gamma'$ because $\Gamma(f_1) \neq \Gamma'(f_1)$, that is $\text{LC}^{n>0} \neq \text{LC}^{n-1}$. We also know that $\delta \neq \delta'$ because $\delta(f_1) \neq \delta'(f_1)$, that is $\text{LC}^{n>0} \neq \text{LC}^{n-1}$. Since $\Gamma \neq \Gamma'$ because $\Gamma(f_1) \neq \Gamma'(f_1)$, and $\delta \neq \delta'$ because $\delta(f_1) \neq \delta'(f_1)$, and $\Gamma'(f_1) = \delta'(f_1)$, that is $\text{LC}^{n-1} = \text{LC}^{n-1}$ and $\Gamma = \delta$. Then, $\Gamma' = \delta'$ as required.

(d) $: \Gamma \mid \Gamma' \vdash f_1 : \text{UC} \quad \Gamma' \mid \Gamma'' \vdash f_2 : \text{UC}$

In this case, the typing derivation of e must have the form $\Gamma \mid \Gamma'', f_1 : \text{UC}, f_2 : \text{UC} \vdash \texttt{cp}\ f_1\ f_2 : \text{void}$. Let $\Gamma' = \Gamma'', f_1 : \text{UC}, f_2 : \text{UC}$. Now we know that $\Gamma = \Gamma'$ and $\delta = \delta'$. Since $\Gamma = \Gamma'$ and $\delta = \delta'$ and $\Gamma = \delta$, then $\Gamma' = \delta'$.

(e) $: \Gamma \mid \Gamma' \vdash f_1 : \text{UC} \quad \Gamma' \mid \Gamma'' \vdash f_2 : \text{LC}^{n>0}$

In this case, the typing derivation of e must have the form $\Gamma \mid \Gamma'', f_1 : \text{UC}, f_2 : \text{LC}^{n>0} \vdash \texttt{cp}\ f_1\ f_2 : \text{void}$. Let $\Gamma' = \Gamma'', f_1 : \text{UC}, f_2 : \text{LC}^{n>0}$. Now we know that $\Gamma = \Gamma'$ and $\delta = \delta'$. Since $\Gamma = \Gamma'$ and $\delta = \delta'$ and $\Gamma = \delta$, then $\Gamma' = \delta'$.

(f) $: \Gamma \mid \Gamma' \vdash f_1 : \text{UC} \quad \Gamma' \mid \Gamma'' \vdash f_2 : \text{NC}$

In this case, the typing derivation of e must have the form $\Gamma \mid \Gamma'', f_1 : \text{UC}, f_2 : \text{NC} \vdash \texttt{cp}\ f_1\ f_2 : \text{void}$. Let $\Gamma' = \Gamma'', f_1 : \text{UC}, f_2 : \text{NC}$. Now we know that $\Gamma = \Gamma'$ and $\delta = \delta'$. Since $\Gamma = \Gamma'$ and $\delta = \delta'$ and $\Gamma = \delta$, then $\Gamma' = \delta'$.

2. $e = \langle \texttt{rm}\ f, \delta \rangle \rightarrow \delta[-f]$

We know there is a typing derivation for e by using rule ($\texttt{rm}$) with conclusion: $\Gamma \mid \Gamma' \vdash \texttt{rm}\ f : \text{void}$. We must also have subderivation with conclusion: $\Gamma \mid \Gamma' \vdash f : \ \tau$. We know that $\Gamma \neq \Gamma'$ because $f \notin \Gamma'$. We also know that $\delta \neq \delta[-f]$ because $f \notin \delta[-f]$. Since $\Gamma \neq \Gamma'$ because $f \notin \Gamma'$ and $\delta \neq \delta[-f]$ because $f \notin \delta[-f]$, and $\Gamma = \delta$. Then, $\Gamma' = \delta[-f]$ as required.

3. $e = \langle \texttt{mkf}\ f\ t, \delta \rangle \rightarrow \delta[+f][f \leftarrow t]$

We know there is a typing derivation for e by using rule ($\texttt{mkf}$) with conclusion: $\Gamma \mid \Gamma, f : t \vdash \texttt{mkf}\ f\ t : \text{void}$. Let $\Gamma' = \Gamma, f : t$ and $\delta' = \delta[+f][f \leftarrow t]$. We know that

$\Gamma \neq \Gamma'$ because $f : t \in \Gamma'$. We also know that $\delta \neq \delta'$ because $f : t \in \delta'$. Since $\Gamma \neq \Gamma'$ because $f : t \in \Gamma'$, and $\delta \neq \delta'$ because $f : t \in \delta'$, and $\Gamma'(f) = \delta'(f)$, and $\Gamma = \delta$. Then, $\Gamma' = \delta'$ as required.

4. $e = \langle \texttt{rd}\ f, \delta\rangle \rightarrow \delta[-f]$

   We know there is a typing derivation for e by using rule ($\texttt{rd}$) with conclusion: $\Gamma \mid \Gamma' \vdash \texttt{rd}\ f : \text{void}$. We must also have subderivation with conclusion: $\Gamma \mid \Gamma' \vdash f : \tau$. We know that $\Gamma \neq \Gamma'$ because $f \notin \Gamma'$. We also know that $\delta \neq \delta[-f]$ because $f \notin \delta[-f]$. Since $\Gamma \neq \Gamma'$ because $f \notin \Gamma'$ and $\delta \neq \delta[-f]$ because $f \notin \delta[-f]$, and $\Gamma = \delta$. Then, $\Gamma' = \delta[-f]$ as required.

5. $e = \langle \texttt{cat}\ f_1\ f_2\ f_3, \delta\rangle \rightarrow \delta[f_3 \leftarrow C(f_1) + C(f_2)][f_3 \leftarrow T(f_1) \sqcup T(f_2) \sqcup T(f_3)][-f_1, -f_2]$

   We know there is a typing derivation for e by using rule ($\texttt{cat}$) with conclusion: $\Gamma \mid \Gamma''', f_3 : \tau \sqcup \tau' \sqcup \tau'' \vdash \texttt{cat}\ f_1\ f_2\ f_3 : \text{void}$. We must also have subderivations with conclusions: $\Gamma \mid \Gamma' \vdash f_1 : \tau$, $\Gamma' \mid \Gamma'' \vdash f_2 : \tau'$, and $\Gamma'' \mid \Gamma''' \vdash f_3 : \tau''$. Let $\delta' = \delta[f_3 \leftarrow C(f_1) + C(f_2)][f_3 \leftarrow T(f_1) \sqcup T(f_2) \sqcup T(f_3)][-f_1, -f_2]$. Now we have the following cases based on typing f_1, f_2, and f_3.

   (a) $\Gamma \mid \Gamma' \vdash f_1 : \text{NC} \quad \Gamma' \mid \Gamma'' \vdash f_2 : \text{LC}^{n>0} \quad \Gamma'' \mid \Gamma''' \vdash f_3 : \text{UC}$

   In this case, the typing derivation of e must have the form $\Gamma \mid \Gamma''', f_3 : \text{NC} \vdash \texttt{cat}\ f_1\ f_2\ f_3 : \text{void}$. Let $\Gamma' = \Gamma''', f_3 : \text{NC}$. We know that $\Gamma \neq \Gamma'$ because $f_1 : \text{NC} \notin \Gamma'$, $f_2 : \text{LC}^{n>0} \notin \Gamma'$ and $\Gamma(f_3) \neq \Gamma'(f_3)$, that is $\text{UC} \neq \text{NC}$. We also know that $\delta \neq \delta'$ because $f_1 : \text{NC} \notin \delta'$, $f_2 : \text{LC}^{n>0} \notin \delta'$ and $\delta(f_3) \neq \delta'(f_3)$, that is $\text{UC} \neq \text{NC}$. Since $\Gamma \neq \Gamma'$ because $f_1 : \text{NC} \notin \Gamma'$, $f_2 : \text{LC}^{n>0} \notin \Gamma'$ and $\Gamma(f_3) \neq \Gamma'(f_3)$, and $\delta \neq \delta'$ because $f_1 : \text{NC} \notin \delta'$, $f_2 : \text{LC}^{n>0} \notin \delta'$ and $\delta(f_3) \neq \delta'(f_3)$, and $\Gamma'(f_3) = \delta'(f_3)$, that is $\text{NC} = \text{NC}$, and $\Gamma = \delta$. Then, $\Gamma' = \delta'$.

   Other cases are similar.

6. $e = \langle \texttt{mv}\ f_1\ f_2\ , \delta\ \rangle \rightarrow \delta[f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_1) \sqcup T(f_2)][-f_1]$

   We know there is a typing derivation for e by using rule ($\texttt{mv}$) with conclusion: $\Gamma \mid \Gamma'', f_2 : \tau \sqcup \tau' \vdash \texttt{mv}\ f_1\ f_2 : \text{void}$. We must also have subderivations with conclusions: $\Gamma \mid \Gamma' \vdash f_1 : \tau$ and $\Gamma' \mid \Gamma'' \vdash f_2 : \tau'$. Let $\delta' = \delta[f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_1) \sqcup T(f_2)][-f_1]$. Now we have the following cases based on typing f_1 and f_2.

   (a) $\Gamma \mid \Gamma' \vdash f_1 : \text{NC} \quad \Gamma' \mid \Gamma'' \vdash f_2 : \text{UC}$

In this case, the typing derivation of e must have the form $\Gamma \mid \Gamma'', f_2 : \text{NC} \vdash$ `mv` f_1 f_2 : void. Let $\Gamma' = \Gamma'', f_2 : \text{NC}$. Now we know that $\Gamma \neq \Gamma'$ because $f_1 : \text{NC} \notin \Gamma'$ and $\Gamma(f_2) \neq \Gamma'(f_2)$, that is UC $\neq$ NC. We also know that $\delta \neq \delta'$ because $f_1 : \text{NC} \notin \delta'$ and $\delta(f_2) \neq \delta'(f_2)$, that is UC $\neq$ NC. Since $\Gamma \neq \Gamma'$ because $f_1 : \text{NC} \notin \Gamma'$ and $\Gamma(f_2) \neq \Gamma'(f_2)$, and $\delta \neq \delta'$ because $f_1 : \text{NC} \notin \delta'$ and $\delta(f_2) \neq \delta'(f_2)$, and $\Gamma'(f_2) = \delta'(f_2)$, and $\Gamma = \delta$. Then, $\Gamma' = \delta'$.

Other cases are similar.

7. $e = \langle$`copy` f_1 f_2 $, \delta\rangle \rightarrow \delta[+f_2, f_2 \leftarrow C(f_1)][f_2 \leftarrow \mathit{dst}(T(f_1))][f_1 \leftarrow \mathit{red}(T(f_1))]$

We know there is a typing derivation for e by using rule (`copy`) with conclusion: $\Gamma \mid \Gamma', f_1 : \mathit{red}(\tau), f_2 : \mathit{dst}(\tau) \vdash$ `copy` f_1 f_2 : void. We must also have subderivation with conclusion: $\Gamma \mid \Gamma' \vdash f_1 : \tau$ and $\tau \sqsubseteq \text{LC}^{n>0}$. Let $\delta' = \delta[+f_2, f_2 \leftarrow C(f_1)][f_2 \leftarrow \mathit{dst}(T(f_1))][f_1 \leftarrow \mathit{red}(T(f_1))]$. Now we have the following cases based on typing f_1.

(a) $\Gamma \mid \Gamma' \vdash f_1 : \text{LC}^{n>0}$

In this case, the typing derivation of e must have the form $\Gamma \mid \Gamma'', f_1 : \text{LC}^{n-1},\ f_2 : \text{NC} \vdash$ `cp` f_1 f_2 : void. Let $\Gamma' = \Gamma'', f_1 : \text{LC}^{n-1},\ f_2 : \text{NC}$. We know that $\Gamma \neq \Gamma'$ because $\Gamma(f_1) \neq \Gamma'(f_1)$, that is $\text{LC}^{n>0} \neq \text{LC}^{n-1}$ and $f_2 : \text{NC} \in \Gamma'$. We also know that $\delta \neq \delta'$ because $\delta(f_1) \neq \delta'(f_1)$, that is $\text{LC}^{n>0} \neq \text{LC}^{n-1}$ and $f_2 : \text{NC} \in \delta'$. Since $\Gamma \neq \Gamma'$ because $\Gamma(f_1) \neq \Gamma'(f_1)$ and $f_2 : \text{NC} \in \Gamma'$, and $\delta \neq \delta'$ because $\delta(f_1) \neq \delta'(f_1)$ and $f_2 : \text{NC} \in \delta'$, and $\Gamma'(f_1) = \delta'(f_1)$, that is $\text{LC}^{n-1} = \text{LC}^{n-1}$, and $\Gamma'(f_2) = \delta'(f_2)$, that is NC $=$ NC, and $\Gamma = \delta$. Then, $\Gamma' = \delta'$.

(b) $\Gamma \mid \Gamma' \vdash f_1 : \text{UC}$

In this case, the typing derivation of e must have the form $\Gamma \mid \Gamma'', f_1 : \text{UC},\ f_2 : \text{UC} \vdash$ `cp` f_1 f_2 : void. Let $\Gamma' = \Gamma'', f_1 : \text{UC},\ f_2 : \text{UC}$. We know that $\Gamma \neq \Gamma'$ because $f_2 : \text{UC} \in \Gamma'$. We also know that $\delta \neq \delta'$ because $f_2 : \text{UC} \in \delta'$. Since $\Gamma \neq \Gamma'$ because $f_2 : \text{UC} \in \Gamma'$, and $\delta \neq \delta'$ because $f_2 : \text{UC} \in \delta'$, and $\Gamma'(f_2) = \delta'(f_2)$, that is UC $=$ UC, and $\Gamma = \delta$. Then, $\Gamma' = \delta'$.

8. $e = \langle$`append` $f_1\ f_2\ f_3, \delta\rangle \rightarrow \delta[+f_3, f_3 \leftarrow C(f_1) + C(f_2)][f_3 \leftarrow T(f_1) \sqcup T(f_2)][-f_1, -f_2]$

We know there is a typing derivation for e by using rule (`append`) with conclusion: $\Gamma \mid \Gamma'', f_3 : \tau \sqcup \tau' \vdash$ `append` f_1 f_2 f_3 : void. We must also have subderivations with conclusions: $\Gamma \mid \Gamma' \vdash f_1 : \tau$ and $\Gamma' \mid \Gamma'' \vdash f_2 : \tau'$. Let $\delta' = \delta[+f_3, f_3 \leftarrow C(f_1) + C(f_2)][f_3 \leftarrow T(f_1) \sqcup T(f_2)][-f_1, -f_2]$. Now we have 6 cases based on typing f_1 and f_2, we show two of them.

(a) $: \Gamma \mid \Gamma' \vdash f_1 : \text{UC} \;\; \Gamma' \mid \Gamma'' \vdash f_2 : \text{NC}$

In this case, the typing derivation of e must have the form $\Gamma \mid \Gamma'', f_3 : \text{NC} \vdash$ `append` $f_1\ f_2\ f_3 : \text{void}$. Let $\Gamma' = \Gamma'', f_3 : \text{NC}$. We know that $\Gamma \neq \Gamma'$ because $f_1 : \text{UC} \notin \Gamma'$, $f_2 : \text{NC} \notin \Gamma'$, and $f_3 : \text{NC} \in \Gamma'$. We also know that $\delta \neq \delta'$ because $f_1 : \text{UC} \notin \delta'$, $f_2 : \text{NC} \notin \delta'$, and $f_3 : \text{NC} \in \delta'$. Since $\Gamma \neq \Gamma'$ because $f_1 : \text{UC} \notin \Gamma'$, $f_2 : \text{NC} \notin \Gamma'$ and $f_3 : \text{NC} \in \Gamma'$, and $\delta \neq \delta'$ because $f_1 : \text{UC} \notin \delta'$, $f_2 : \text{NC} \notin \delta'$ and $f_3 : \text{NC} \in \delta'$, and $\Gamma = \delta$. Then, $\Gamma' = \delta'$.

(b) $: \Gamma \mid \Gamma' \vdash f_1 : \text{UC} \;\; \Gamma' \mid \Gamma'' \vdash f_2 : \text{LC}^{n>0}$

In this case, the typing derivation of e must have the form $\Gamma \mid \Gamma'', f_3 : \text{LC}^{n>0} \vdash$ `append` $f_1\ f_2\ f_3 : \text{void}$. Let $\Gamma' = \Gamma'', f_3 : \text{LC}^{n>0}$. We know that $\Gamma \neq \Gamma'$ because $f_1 : \text{UC} \notin \Gamma'$, $f_2 : \text{LC}^{n>0} \notin \Gamma'$, and $f_3 : \text{LC}^{n>0} \in \Gamma'$. We also know that $\delta \neq \delta'$ because $f_1 : \text{UC} \notin \delta'$, $f_2 : \text{LC}^{n>0} \notin \delta'$, and $f_3 : \text{LC}^{n>0} \in \delta'$. Since $\Gamma \neq \Gamma'$ because $f_1 : \text{UC} \notin \Gamma'$, $f_2 : \text{LC}^{n>0} \notin \Gamma'$ and $f_3 : \text{LC}^{n>0} \in \Gamma'$, and $\delta \neq \delta'$ because $f_1 : \text{UC} \notin \delta'$, $f_2 : \text{LC}^{n>0} \notin \delta'$ and $f_3 : \text{LC}^{n>0} \in \delta'$, and $\Gamma = \delta$. Then, $\Gamma' = \delta'$.

Other cases are similar.

9. $e = \langle$`move` $f_1\ f_2, \delta\ \rangle \to \delta[+f_2, f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_1)][-f_1]$

We know there is a typing derivation for e by using rule (`move`) with conclusion: $\Gamma \mid \Gamma', f_2 : \tau \vdash$ `move` $f_1\ f_2 : \text{void}$. We must also have subderivation with conclusion: $\Gamma \mid \Gamma' \vdash f_1 : \tau$. Let $\delta' = \delta[+f_2, f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_1)][-f_1]$. Now we have the following cases based on typing f_1.

(a) $\Gamma \mid \Gamma' \vdash f_1 : \text{NC}$

In this case, the typing derivation of e must have the form $\Gamma \mid \Gamma', f_2 : \text{NC} \vdash$ `move` $f_1\ f_2 : \text{void}$. Let $\Gamma' = \Gamma', f_2 : \text{NC}$. Now we know that $\Gamma \neq \Gamma'$ because $f_1 : \text{NC} \notin \Gamma'$ and $f_2 : \text{NC} \in \Gamma'$. We also know that $\delta \neq \delta'$ because $f_1 : \text{NC} \notin \delta'$ and $f_2 : \text{NC} \in \delta'$. Since $\Gamma \neq \Gamma'$ because $f_1 : \text{NC} \notin \Gamma'$ and $f_2 : \text{NC} \in \Gamma'$, and $\delta \neq \delta'$ because $f_1 : \text{NC} \notin \delta'$ and $f_2 : \text{NC} \in \delta'$, and $\Gamma'(f_2) = \delta'(f_2)$, and $\Gamma = \delta$. Then, $\Gamma' = \delta'$.

Other cases are similar.

$\square$

5.9 Type inference algorithm

In this section we present a type inference algorithm $\mathcal{T}$ for typing phrases. The type inference algorithm $\mathcal{T}$ finds the type of a phrase within a given type environment if any such type exists. That is, for a given phrase e and initial type environment A, the type inference algorithm computes a type τ of e if e is typable or fails if e is not typable. We prove two properties of the algorithm $\mathcal{T}$ which are *soundness* and *completeness*. The algorithm $\mathcal{T}$ is sound if it only computes types of phrases that are typable in the given type environment. This property is useful to show that the algorithm $\mathcal{T}$ will not give false positive results. The algorithm $\mathcal{T}$ is complete if it only fails to compute types of phrases that are not typable in the given type environment. This property is useful to show that the algorithm $\mathcal{T}$ will not give false negative results.

5.9.1 Algorithm $\mathcal{T}$

The algorithm $\mathcal{T}$ is defined as a recursive function. The main function $\mathcal{T}(A,e)$, takes a type environment A and a phrase e, and computes (τ, A') which is the unique type τ of e and the new environment A'. The algorithm fails if any of the recursive invocations of $\mathcal{T}(A,e)$ fails or any of the invocations of the helper functions defined below fails. Such failure indicates a typing error. We define a number of helper functions: $\mathit{check}(\alpha,\beta)$ returns true if the types are compatible. Note that any two base types are not compatible, e.g. $\mathit{check}(\mathit{LC},\text{void})$ will fail, and type variable and base type will always succeed. $\mathit{less}(\tau,\tau')$ returns true if $\tau \sqsubseteq \tau'$. $\mathit{lub}(\tau,\ldots,\tau_n)$ returns the least upper bound of all its parameters i.e. $\tau \sqcup \ldots \sqcup \tau_n$. It should be noted that unification is not needed in our case, since we need to check for base types rather variable types. Using these functions, we can now define the type inference algorithm $\mathcal{T}$:

$$\mathcal{T}(A,e) = (\tau, A')$$

where:

1. If e is the filename f, and $f:\alpha \in A$ then $\tau = \alpha$, $A' = A \setminus \{f:\alpha\}$.

2. If e is a sequence of commands, $c; cs$ let

$$\begin{array}{lcl} (\beta, A_1) & = & \mathcal{T}(A,c) \\ \mathit{check}(\beta,\text{void}) & & \\ (\alpha, A_2) & = & \mathcal{T}(A_1,cs) \\ \mathit{check}(\alpha,\text{void}) & & \end{array}$$

   then $\tau = \text{void}, A' = A_2$.

3. If e is the command `cp` f_1 f_2 let

$$\begin{array}{lcl} (\beta, A_1) & = & \mathcal{T}(A, f_1) \\ less(\beta, LC^{n>0}) & & \\ (\alpha, A_2) & = & \mathcal{T}(A_1, f_2) \end{array}$$

   then if $f_1, f_2 \notin A_2$, then $\tau =$ void, $A' = A_2 \cup \{f_1 : \mathit{red}(\beta), f_2 : \mathit{lub}(\alpha, \mathit{dst}(\beta))\}$.

4. If e is the command `rm` f let

$$(\alpha, A_1) = \mathcal{T}(A, f)$$

   then $\tau =$ void, $A' = A_1$.

5. If e is the command `mkf` f t, then if $f \notin A$, then $\tau =$ void, $A' = A \cup \{f : t\}$.

6. If e is the command `rd` f let

$$(\alpha, A_1) = \mathcal{T}(A, f)$$

   then $\tau =$ void, $A' = A_1$.

7. If e is the command `cat` f_1 f_2 f_3 let

$$\begin{array}{lcl} (\beta, A_1) & = & \mathcal{T}(A, f_1) \\ (\alpha, A_2) & = & \mathcal{T}(A_1, f_2) \\ (\delta, A_3) & = & \mathcal{T}(A_2, f_3) \end{array}$$

   then if $f_3 \notin A_3$, then $\tau =$ void, $A' = A_3 \cup \{f_3 : \mathit{lub}(\beta, \alpha, \delta)\}$.

8. If e is the command `mv` f_1 f_2 let

$$\begin{array}{lcl} (\beta, A_1) & = & \mathcal{T}(A, f_1) \\ (\alpha, A_2) & = & \mathcal{T}(A_1, f_2) \end{array}$$

   then if $f_2 \notin A_2$, then $\tau =$ void, $A' = A_2 \cup \{f_2 : \mathit{lub}(\beta, \alpha)\}$.

9. If e is the command `copy` f_1 f_2 let

$$\begin{array}{lcl} (\beta, A_1) & = & \mathcal{T}(A, f_1) \\ less(\beta, LC^{n>0}) & & \end{array}$$

   then if $f_2 \notin A$ and $f_1 \notin A_1$, then $\tau =$ void, $A' = A_1 \cup \{f_1 : \mathit{red}(\beta), f_2 : \mathit{dst}(\beta)\}$.

10. If e is the command `append` f_1 f_2 f_3 let

$$\begin{aligned}(\beta, A_1) &= \mathcal{T}(A, f_1)\\ (\alpha, A_2) &= \mathcal{T}(A_1, f_2)\end{aligned}$$

    then if $f_3 \notin A$, then $\tau = \text{void}, A' = A_2 \cup \{f_3 : \mathit{lub}(\beta, \alpha)\}$.

11. If e is the move command, `move` f_1 f_2 let

$$(\beta, A_1) = \mathcal{T}(A, f_1)$$

    then if $f_2 \notin A$, then $\tau = \text{void}, A' = A_1 \cup \{f_2 : \beta\}$.

5.9.2 Soundness of $\mathcal{T}$

In this section we show that the algorithm $\mathcal{T}$ is sound with respect to the formal typing system. The algorithm $\mathcal{T}$ is sound if for any given type environment A and phrase e, if the algorithm $\mathcal{T}$ on input A and e computes the new environment A' and the type τ, then there is a derivation for e such that $A \mid A' \vdash e : \tau$ in the type system:

$$\mathcal{T}(A, e) = (\tau, A') \Rightarrow A \mid A' \vdash e : \tau$$

Theorem 5.9.1 (Soundness of $\mathcal{T}$)**.** If $\mathcal{T}(A, e)$ succeeds with (τ, A'), then there is a derivation ending in $A \mid A' \vdash e : \tau$.

Proof. We proceed by induction on the structure of the prhase e. There are 11 cases as follows.

1. If e is the filename f and $f : \tau \in A \cup \{f : \tau\}$, then $\mathcal{T}(A \cup \{f : \tau\}, f)$ succeeds immediately with (τ, A). Using the (f) rule, there is a derivation ending in $A, f : \tau \mid A \vdash f : \tau$ as required.

2. If e is the sequence of commands $c; cs$, then $\mathcal{T}(A, c)$ succeeds with (β, A_1), $\mathit{check}(\beta, \text{void})$ succeeds, $\mathcal{T}(A_1, cs)$ succeeds with (α, A_2), and $\mathit{check}(\alpha, \text{void})$ also succeeds. Now, by the inductive hypothesis twice, there are derivations $A \mid A_1 \vdash c : \text{void}$ and $A_1 \mid A_2 \vdash cs : \text{void}$. Using the (cs) rule, there is a derivation $A \mid A_2 \vdash c; cs : \text{void}$ as required.

3. If e is the cp command `cp` f_1 f_2, then $\mathcal{T}(A, f_1)$ succeeds with (β, A_1) and $\mathcal{T}(A_1, f_2)$ succeeds with (α, A_2). By the inductive hypothesis twice, there are derivations ending in $A \mid A_1 \vdash f_1 : \beta$ and $A_1 \mid A_2 \vdash f_2 : \alpha$. Since $\mathit{less}(\beta, \mathrm{LC}^{n>0})$, we have $\beta \sqsubseteq \mathrm{LC}^{n>0}$ and now we can use the (`cp`) rule to give a derivation of $A \mid A_2, f_1 : \mathit{red}(\beta), f_2 : \alpha \sqcup \mathit{dst}(\beta) \vdash$ `cp` f_1 f_2 : void as required.

4. If e is the rm command `rm` f, then $\mathcal{T}(A, f)$ succeeds with (α, A_1). By the inductive hypothesis, there is a derivation of $A \mid A_1 \vdash f : \alpha$. Now we can use the (`rm`) rule to give a derivation of $A \mid A_1 \vdash \texttt{rm}\ f : \text{void}$ as required.

5. If e is the mkf command `mkf` $f\ t$ and $f \notin A$, then $\mathcal{T}(A, (mkf\ f\ t))$ succeeds immediately with $(\text{void}, A \cup \{f : \tau\})$. Using the (`mkf`) rule, there is a derivation $A \mid A, f : t \vdash \texttt{mkf}\ f\ t : \text{void}$ as required.

6. If e is the rd command `rd` f, then $\mathcal{T}(A, f)$ succeeds with (α, A_1). By the inductive hypothesis, there is a derivation ending in $A \mid A_1 \vdash f : \alpha$. Using the (`rd`) rule, there is a derivation $A \mid A_1 \vdash \texttt{rd}\ f : \text{void}$ as required.

7. If e is the cat command `cat` $f_1\ f_2\ f_3$, then $\mathcal{T}(A, f_1)$ succeeds with (β, A_1), $\mathcal{T}(A_1, f_2)$ succeeds with (α, A_2), and $\mathcal{T}(A_2, f_3)$ succeeds with (δ, A_3). Now, by the inductive hypothesis triple, there are derivations ending in $A \mid A_1 \vdash f_1 : \beta,\ \ A_1 \mid A_2 \vdash f_2 : \alpha$, and $A_2 \mid A_3 \vdash f_3 : \delta$. Using the (`cat`) rule, there is a derivation $A \mid A_3, f_3 : \beta \sqcup \alpha \sqcup \delta \vdash \texttt{cat}\ f_1\ f_2\ f_3 : \text{void}$ as required.

8. If e is the mv command `mv` $f_1\ f_2$, then $\mathcal{T}(A, f_1)$ succeeds with (β, A_1) and $\mathcal{T}(A_1, f_2)$ succeeds with (α, A_2). By the inductive hypothesis twice, there are derivations ending in $A \mid A_1 \vdash f_1 : \beta$ and $A_1 \mid A_2 \vdash f_2 : \alpha$. Now we can use the (`mv`) rule to give a derivation of $A \mid A_2, f_2 : \beta \sqcup \alpha \vdash \texttt{mv}\ f_1\ f_2 : \text{void}$ as required.

9. If e is the copy command `copy` $f_1\ f_2$, then $\mathcal{T}(A, f_1)$ succeeds with (β, A_1). By the inductive hypothesis, there is a derivation ending in $A \mid A_1 \vdash f_1 : \beta$. Since $less(\beta, \text{LC}^{n>0})$, we have $\beta \sqsubseteq \text{LC}^{n>0}$ and now we can use the (`copy`) rule to give a derivation of $A \mid A_1, f_1 : red(\beta) f_2 : dst(\beta) \vdash \texttt{copy}\ f_1\ f_2 : \text{void}$ as required.

10. If e is the append command `append` $f_1\ f_2\ f_3$, then $\mathcal{T}(A, f_1)$ succeeds with (β, A_1) and $\mathcal{T}(A_1, f_2)$ succeeds with (α, A_2). By the inductive hypothesis twice, there are derivations ending in $A \mid A_1 \vdash f_1 : \beta$ and $A_1 \mid A_2 \vdash f_2 : \alpha$. Now by using the (`append`) rule, there is a derivation of $A \mid A_2, f_3 : \beta \sqcup \alpha \vdash \texttt{append}\ f_1\ f_2\ f_3 : \text{void}$ as required.

11. If e is the move command `move` $f_1\ f_2$, then $\mathcal{T}(A, f_1)$ succeeds with (β, A_1). By the inductive hypothesis, there is a derivation ending in $A \mid A_1 \vdash f_1 : \beta$. Using the (`move`) rule, there is a derivation of $A \mid A_1, f_2 : \beta \vdash \texttt{move}\ f_1\ f_2 : \text{void}$ as required.

$\square$

5.9.3 Completeness of $\mathcal{T}$

In this section we show that the algorithm $\mathcal{T}$ is complete with respect to the formal typing system. The algorithm $\mathcal{T}$ is complete if for any given type environment A and phrase e, if there is a derivation for e such that $A \mid A' \vdash e : \tau$ in the type system, then the algorithm $\mathcal{T}$ on input A and e will compute the new environment A' and the type τ.

$$A \mid A' \vdash e : \tau \Rightarrow \mathcal{T}(A, e) = (\tau, A')$$

Theorem 5.9.2 (Completeness of $\mathcal{T}$)**.** If there is a derivation ending in $A \mid A' \vdash e : \tau$, then $\mathcal{T}(A, e)$ succeeds with (τ, A').

Proof. We proceed by induction on the structure of the prhase e. There are 11 cases as follows.

1. If e is the file name f and $f : \tau \in A \cup \{f : \tau\}$, then by (f) rule, there is a derivation ending in $A, f : \tau \mid A \vdash f : \tau$. Now, $\mathcal{T}(A \cup \{f : \tau\}, f)$ succeeds with (τ, A) as required.

2. If e is the sequence of commands $c; cs$, then by (cs) rule there is a derivation ending in $A \mid A_2 \vdash c; cs : \text{void}$ which consists of two derivations: $A \mid A_1 \vdash c : \text{void}$ and $A_1 \mid A_2 \vdash cs : \text{void}$. By the induction hypothesis twice, $\mathcal{T}(A, c)$ succeeds with (void, A_1) for the first derivation and $\mathcal{T}(A_1, cs)$ succeeds with (void, A_2) for the second derivation. Now $\mathcal{T}(A, (c, cs))$ succeeds with (void, A_2) as requied.

3. If e is the cp command $\texttt{cp}\ f_1\ f_2$, then by $(\texttt{cp})$ rule, there is a derivation ending in $A \mid A_2, f_1 : \mathit{red}(\beta), f_2 : \alpha \sqcup \mathit{dst}(\beta) \vdash \texttt{cp}\ f_1\ f_2 : \text{void}$ which consists of two derivations: $A \mid A_1 \vdash f_1 : \beta$ and $A_1 \mid A_2 \vdash f_2 : \alpha$. By the induction hypothesis twice and since $\beta \sqsubseteq \text{LC}^{n>0}$ we have $\mathit{less}(\beta, \text{LC}^{n>0})$, $\mathcal{T}(A, f_1)$ succeeds with (β, A_1) for the first derivation, and $\mathcal{T}(A_1, f_2)$ succeeds with (α, A_2) for the second derivation. Now $\mathcal{T}(A, (cp\ f_1\ f_2))$ succeeds with $(\text{void}, A_2 \cup \{f_1 : \mathit{red}(\beta), f_2 : \mathit{lub}(\alpha, \mathit{dst}(\beta))\})$ as required.

4. If e is the rm command $\texttt{rm}\ f$, then by $(\texttt{rm})$ rule there is a derivation ending in $A \mid A_1 \vdash \texttt{rm}\ f : \text{void}$ which consists of one derivation: $A \mid A_1 \vdash f : \beta$. By the induction hypothesis, $\mathcal{T}(A, f)$ succeeds with (β, A_1). Now $\mathcal{T}(A, (rm\ f))$ succeeds with (void, A_1) as required.

5. If e is the command $\texttt{mkf}\ f\ t$ and $f \notin A$, then by $(\texttt{mkf})$ rule, there is a derivation ending in $A \mid A, f : \tau \vdash \texttt{mkf}\ f\ t : \text{void}$. Now, $\mathcal{T}(A, (mkf\ f\ t))$ succeeds with $(\text{void}, A \cup \{f : \tau\})$ as required.

6. If e is the command `rd` f, then by (`rd`) rule there is a derivation ending in $A \mid A_1 \vdash$ `rd` f : void which consists of one derivation: $A \mid A_1 \vdash f : \beta$. By the induction hypothesis, $\mathcal{T}(A, f)$ succeeds with (β, A_1). Now, $\mathcal{T}(A, (\texttt{rd}\ f))$ succeeds with (void, A_1) as required.

7. If e is the cat command `cat` $f_1\ f_2\ f_3$, then by (`cat`) rule, there is a derivation ending in $A \mid A_3, f_3 : \beta \sqcup \alpha \sqcup \delta \vdash \texttt{cat}\ f_1\ f_2\ f_3$: void which consists of three derivations: $A \mid A_1 \vdash f_1 : \beta$, $A_1 \mid A_2 \vdash f_2 : \alpha$, and $A_2 \mid A_3 \vdash f_3 : \delta$. By the induction hypothesis triple, $\mathcal{T}(A, f_1)$ succeeds with (β, A_1), $\mathcal{T}(A_1, f_2)$ succeeds with (α, A_2), and $\mathcal{T}(A_2, f_3)$ succeeds with (δ, A_3). Now, $\mathcal{T}(A, (\texttt{cat}\ f_1\ f_2\ f_3))$ succeeds with $(\mathit{void}, A_3 \cup \{f_3 : \mathit{lub}(\beta, \alpha, \delta)\})$ as required.

8. If e is the mv command `mv` $f_1\ f_2$, then by (`mv`) rule, there is a derivation ending in $A \mid A_2, f_2 : \beta \sqcup \alpha \vdash \texttt{mv}\ f_1\ f_2$: void which consists of two derivations: $A \mid A_1 \vdash f_1 : \beta$ and $A_1 \mid A_2 \vdash f_2 : \alpha$. By the induction hypothesis twice, $\mathcal{T}(A, f_1)$ succeeds with (β, A_1) and $\mathcal{T}(A_1, f_2)$ succeeds with (α, A_2). Now, $\mathcal{T}(A, (\texttt{mv}\ f_1\ f_2))$ succeeds with $(\text{void}, A_2 \cup \{f_2 : \mathit{lub}(\beta, \alpha)\})$ as required.

9. If e is the copy command `copy` $f_1\ f_2$, then by (`copy`) rule, there is a derivation ending in $A \mid A_1, f_1 : \mathit{red}(\beta), f_2 : \mathit{dst}(\beta) \vdash \texttt{copy}\ f_1\ f_2$: void which consists of one derivation: $A \mid A_1 \vdash f_1 : \beta$. By the induction hypothesis and since $\beta \sqsubseteq \text{LC}^{n>0}$ we have $\mathit{less}(\beta, \text{LC}^{n>0})$, $\mathcal{T}(A, f_1)$ succeeds with (β, A_1). Now $\mathcal{T}(A, (\texttt{copy}\ f_1\ f_2))$ succeeds with $(\text{void}, A_1 \cup \{f_1 : \mathit{red}(\beta) f_2 : \mathit{dst}(\beta)\})$ as required.

10. If e is the append command `append` $f_1\ f_2\ f_3$, then by (`append`) rule, there is a derivation ending in $A \mid A_2, f_3 : \beta \sqcup \alpha \vdash \texttt{append}\ f_1\ f_2\ f_3$: void which consists of two derivations: $A \mid A_1 \vdash f_1 : \beta$ and $A_1 \mid A_2 \vdash f_2 : \alpha$. By the induction hypothesis twice, $\mathcal{T}(A, f_1)$ succeeds with (β, A_1) and $\mathcal{T}(A_1, f_2)$ succeeds with (α, A_2). Now, $\mathcal{T}(A, (\mathit{append}\ f_1\ f_2\ f_3))$ succeeds with $(\text{void}, A_2 \cup \{f_3 : \mathit{lub}(\beta, \alpha)\})$ as required.

11. If e is the move command `move` $f_1\ f_2$, then by (`move`) rule, there is a derivation ending in $A \mid A_1, f_2 : \beta \vdash \texttt{move}\ f_1\ f_2$: void which consists of one derivation: $A \mid A_1 \vdash f_1 : \beta$. By the induction hypothesis, $\mathcal{T}(A, f_1)$ succeeds with (β, A_1). Now, $\mathcal{T}(A, (\texttt{move}\ f_1\ f_2))$ succeeds with $(\text{void}, A_1 \cup \{f_2 : \beta\})$ as required.

$\square$

5.10 Summary

In this chapter we presented our approach to enforce a particular constraint of the policies identified in Chapter 4; namely, limiting the number of times a file can be read. We enforced this constraint by limiting the number of copies a file can produce. We achieved this by defining security types that regulate *copy* operations on files. These security types control the access to *copy* operations and the flow caused by all operations including *copy*, such that policies for copying files are not violated. The file system we consider is based on the notion of resource consumption, that is a file is a resource which is consumed when it is used, unless the file is explicitly copied. Therefore, a file can be read as much as it can be copied. If the file cannot be copied, then it can be read only once. We designed a language of commands to enforce these policies in a file system. The commands of the language can be issued to manipulate files according to their policies. We showed that these commands might result in execution errors. We divided these errors into syntactical and type errors. The former occur when the constraints of an operation applied to the file system are not satisfied while the latter occur when the constraints of an operation applied to a type of a file are not satisfied. We discussed syntactical errors and define an algorithm to check for syntactical correctness of commands before execution. While this suffices to prevent syntactical errors before execution, type errors might still occur. Therefore, we developed a type system that enforces the policies of files and prevents both syntactical errors and type errors of commands before execution. That is, a type-checked commands are guaranteed to not cause errors during execution. We proved the soundness of the type system with respect to the operational semantics of the language. Finally, we define a type inference algorithm for typing phrases in our language, and proved its soundness and completeness.

The type system developed in this chapter can be thought of as a reference monitor that intercepts each command to be performed on files and checks if the command is allowed by the types of the files and enforces the flow policies of these files. Types of files are not necessarily stored with file names and contents in the file system δ. They can be separated from δ and stored in a different location (e.g., Δ) and fetched upon request by the type system. For example, δ will be the set of file names with contents (e.g., $\{f_1(c_1), \ldots, f_n(c_n)\}$) and Δ will be the set of file names with types (e.g., $\{f_1 : \tau_1, \ldots, f_n : \tau_n\}$). For checking commands that need to be executed, the type system makes Δ to be the typing context to begin with. Once all the commands are type-checked correctly, the resulting typing context after the checking (e.g., Γ') should replace the types of files stored in Δ. In this way, we could have an untyped operational semantics that relies solely on the safety guarantee given by the type system. In fact, the reason for having typed operational semantics is just to simplify the soundness proof of the type system—once we have established this result, we can optimise these out.

Unlike conventional security type systems where security types only represent information flow policies, our security types can be thought of as files permissions that represent both access control and information flow policies. They represent access control policies as they dictate which commands can be issued on which types of files. They also represent information flow policies as they form a lattice structure which dictates where information is allowed to flow. For example, information flow from a file f_1 to f_2 is allowed if and only if the permission of f_2 is at least as restrictive as the permission of f_1. Since flow of information can only be caused by issuing commands on files, such commands are subject to access control checks by the type system as well as information flow checks. That is, the type system enforces access control policies by checking whether or not the commands issued on files are allowed by their types, and enforces information flow policies by checking whether or not the information flow caused by the commands, if any exists, satisfies the lattice structure of the types associated with the files. Although the security types presented in this chapter only regulates *copy* operations, other operations can be regulated in the same way as we discuss in the next chapter.

The language and the type system presented in this chapter is kept to a minimum to avoid complexity in presenting our approach. Various extensions useful in practice including conditionals, loops, recursion, and variables are left for future work. We aimed to start this line of research with a very simple language with the desired properties and then extending it while ensuring these properties are still preserved. Therefore, we focused on a small set of commands that are essential to perform the activity of file sharing in a multi-user system such as Unix. In future work we aim to extend the language with various features and the type system to enforce different kinds of policies useful in practice.

In this chapter we presented a small set of atomic commands that manipulate files. These atomic commands can be grouped together to form macro commands. Such macro commands can do the functionality of several atomic commands. An example of a useful macro command is that which combines `copy` and `rd` commands. For example if we introduce `read` as a macro command and define it as follows:

$$\texttt{read}\ f = \texttt{copy}\ f\ f'; \texttt{rd}\ f'$$

then, users will be relieved from having to issue `copy` command on a file each time they need to read it to retain a copy of it after reading it. That is, limiting the number of times a file can be read is automatically handled by the macro command `read`. In this case, users will have two options, either issuing `read` command to read a file and retain a copy of it after reading it, or issuing `rd` command to read a file and consume it. Of course, if the file cannot be copied, then it can only be read and consumed by issuing `rd` command.

At this stage, a program can either be a single command or a sequence of commands. In a single command, the type system checks the command before executing it, to find out whether the command will cause an error or not if it is executed. As mentioned above, we can have untyped operational semantics and rely only on the type system such that type-checked commands can only be executed. This is because it is guaranteed that type-checked commands will not cause an error. In this case, therefore, the usefulness of our approach is that a user cannot execute a command that violates files or system policies. For example, the type system will prevent executing a command that removes a file that does not exist, creates a file that already exists, copies a file that must not be copied, or reads a file more than the allowed number of times. This is because such commands violate our policies, and thus, will cause an error if executed.

On the other hand, in a sequence of commands, the type system checks every command in the sequence before executing them, to find out whether any of them will cause an error during execution. In this case, the usefulness of our approach is that a program which consists of a sequence of commands can be either executed as a whole or nothing will be executed. If we rely on typed operational semantics only, then a sequence of commands will not be executed fully if there is a command, for example in the middle of the sequence, causes an error. That is, some commands might be executed and others might not, and thus, the program will not complete its job.

However, our type system will type-check such program which consists of a sequence of commands, and only allow it to be executed if it is guaranteed that the program will complete its job. That is, every command in the sequence will not cause an error during execution. Having a program completing its job or nothing should be done is of great importance in many scenarios. One of these scenarios is a program which consists of a sequence of commands for copying each file in a file system for a backup purpose. If we execute such program without type-checking, and if there is a copy command in the middle of the sequence which copies a file of type NC. Then, this will cause an error and the result of the program will be incomplete, because files will be copied up to the point of the error and not all of them. However, our type system will reject this program because it contains an error, and thus, cannot be executed. Therefore, either the whole backup is completed or nothing should be copied.

In fact, the importance of a program to complete its job applies to any program consists of a sequence of commands which contains commands to be executed mainly for successful execution of other commands. For example, `copy` f_1 f_2;`rd` f_2 should not be executed if the user issuing this sequence of commands is not allowed to read the file. This is because the file f_1 is copied in order to be read, however, if it cannot be read, then it should not be copied in the first place.

Therefore, our type system helps users to execute correct programs that are not only secure but also guaranteed to complete their jobs without any interruption as a result of an error. Once the current language is extended with conditionals, loops, recursion, and variables, type-checking programs will become essential for successful execution.

Chapter 6

Future extension and discussion

6.1 Introduction

The type system presented in the previous chapter is focused on enforcing a particular constraint which is limiting the number of times a file can be read in a file system (shared memory). This is achieved by controlling the access to *copy* operations and the flow of information caused by any operations, such that policies for copying files are not violated. There are a number of ways in which the type system in the previous chapter can be extended to enforce the various policies identified in Chapter 4. For example, the type checker might not only enforce the number of times a file can be read, but also the different types of access and propagation identified in Chapter 4. In this chapter we investigate some of these possible extensions. The extensions discussed in this chapter require further investigation and proofs of their properties which is our aim for future work. However, a significant step towards realising these extensions is taken in this chapter.

This chapter is organised as follows: in Section 6.2 we define security access types that represent security policies to regulate *read* and *write* operations. We discuss these policies and show how the type system in the previous chapter can be used to enforce the new policies with slight modification to the typing rules. In Section 6.3 we extend the type system presented in the previous chapter to enforce both policies that regulate *copy*, *read*, and *write* operations, and extend the typing algorithm for typing phrases according to the revised type system. In Section 6.4 we extend policies of files to include ownership and authorisation information. We present a revised type system that not only enforces which operations can be performed on which types of files, but also which user can perform these operations. In Section 6.5 we extend the language with commands that manipulate policies of files, and present typing rules for these commands. We revise the typing algorithm presented in Section 6.3 for typing phrases according to the typing rules presented in Sections 6.4 and 6.5. We discuss these extensions in Section 6.6 and summarise the chapter in Section 6.7.

6.2 Accessing files

The security policies imposed by the security types described in the previous chapter are only concerned with the *copy* operations. Such security types regulate how the *copy* operation can be performed on the original files as well as the copy version of these files. Although *copy* is a crucial operation such that controlling it will limit the number of times a file can be read in our system, it is not concerned with accessing the files. Accessing the files can be made either by reading or modifying the files which are performed by the two critical operations *read* and *write*, respectively. Similar to *copy*, the operations *read* and *write* must be governed by the security policies to prevent unauthorised access to the files. In this section we investigate additional security types which we refer to as *security access types* that represent security policies to regulate *read* and *write* operations.

6.2.1 Security access types

We define six security access types to control *read* and *write* operations which are NRW, RO, WO^-, WO^+, RW^-, and RW^+ each of which specifies a distinct policy of how *read* and *write* operations can be performed on them. NRW stands for NoReadOrWrite, which means that a file associated with this type cannot be read or written into it. RO stands for ReadOnly, which means that a file associated with this type can only be read but not written into it. WO^- stands for WriteOnly, which means that a file associated with this type can only be written into it by either appending or removing content from it. WO^+ stands for WriteOnly, which means that a file associated with this type can only be written into by appending content but not removing content from it. RW^- stands for ReadWrite, which means that a file associated with this type can be read or written into it by appending or removing content from it. RW^+ stands for ReadWrite, which means that a file associated with this type can be read or written into it by appending content but not removing content from it.

Similar to the copy security types described in Chapter 5, the security access types form a lattice $(\tau, \sqsubseteq)$ where $\tau = \{\text{NRW}, \text{RO}, \text{WO}^-, \text{WO}^+, \text{RW}^-, \text{RW}^+\}$, are partially ordered by $\sqsubseteq$ (see Figure 6.1). NRW and RW^- are the upper bound and the lower bound of the set τ, respectively. The least restrictive type is RW^-, while the most restrictive type is NRW.

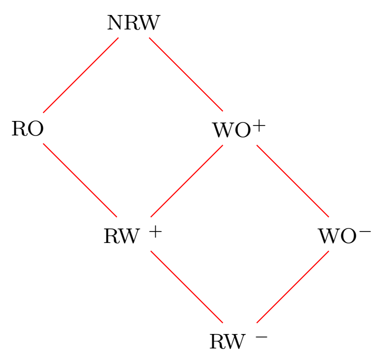

Figure 6.1: Security access types

In the previous chapter we dealt with copy security types and differentiate between flow of information caused by *copy* operations, such as `cp` and `copy`, and by other operations, such as `cat`, `mv`, `append` etc. This is because we added more constraints on performing *copy* operations such as the source file must be of type UC or $\mathrm{LC}^{n>0}$ and the destination file must change its type to be the join of its type and the *dst* of the source type. In this way, we control the access to *copy* operations and the flow caused by all operations including *copy*. In this section we ignore the additional constraints imposed by the *copy* operations and focus on the general policy introduced in the previous chapter and the security access types introduced in this chapter. The policy of information flow generally stated that flow of information from f_1 to f_2 is always allowed, provided that f_2 must change its type to $T(f_1) \sqcup T(f_2)$ and f_1 is consumed after performing the operation, if the operation is not *copy*. Otherwise, f_1 must not be consumed. If $f_2 \notin \mathit{Types}$, then it will be assigned the join of the source types.

This policy violates the meaning of the security access types described above. For example, assume that $\forall f \in \mathit{Types}, T(f) \in \{\mathrm{NRW}, \mathrm{RO}, \mathrm{WO}^-, \mathrm{WO}^+, \mathrm{RW}^-, \mathrm{RW}^+\}$; i.e. that is, types of files are those of security access types only. Now, the typing rules in the previous chapter which reflects the policies described above, violates the policies of the security access types as follows. Assume that $T(f_1) = \mathrm{RO}$ and $T(f_2) = \mathrm{NRW}$, then `mv` f_1 f_2 will result in f_2 changes its type to $T(f_1) \sqcup T(f_2) = \mathrm{NRW}$. Although the new type of f_2 will always be at least as restrictive as both types of f_1 and f_2, the policy of f_2 is violated. This is because the type of f_2 is NRW which requires that such a file cannot be written into it. Now assume that $T(f_1) = \mathrm{NRW}$ and $T(f_2) = \mathrm{WO}^+$, then `cp` f_1 f_2 will result in f_2 changing its type to $T(f_1) \sqcup T(f_2) = \mathrm{NRW}$. Again the policy of f_2 is violated because `cp` f_1 f_2 overwrites the content of f_2 by the content of f_1 whereas the type of f_2 is WO^+ which requires such a file can be written into it by appending but not removing content from it. Finally, assume that $T(f_1) = \text{RO}$, $T(f_2) = \text{RO}$ $T(f_3) = \text{WO}^-$, then `cat` f_1 f_2 f_3 will result in f_3 changing its type to $T(f_1) \sqcup T(f_2) \sqcup T(f_3) = \text{NRW}$. While the policy of f_3 is not violated because its type allows its content to be overwritten, the policies of f_1 and f_2 are violated. This is because `cat` f_1 f_2 f_3 will append f_1 and f_2 together to overwrite f_3, whereas the types of f_1 and f_2 is RO which requires such files to be read only but not written into.

To avoid this violation to the security access types, we need to add constraints to control the access to *write* operations in addition to the flow policies defined in the previous chapter. Thus, we need to enforce the following three policies. *a*) A file f can be read if and only if $T(f) \in \{\text{RW}^-, \text{RW}^+, \text{RO}\}$. *b*) A file f can be overwritten if and only if $T(f) \in \{\text{RW}^-, \text{WO}^-\}$. *c*) A file f can be appended to it if and only if $T(f) \in \{\text{RW}^-, \text{RW}^+\text{WO}^-, \text{WO}^+\}$. In the next section we present the language and the typing rules to enforce the policies described above.

6.2.2 Language and typing rules

The syntax of the language will be the same as described in the previous chapter. However, the only difference is in the types which represent the security access types only. That is,

$$\langle t \rangle \quad ::= \quad \text{NRW} \mid \text{RO} \mid \text{WO}^- \mid \text{WO}^+ \mid \text{RW}^- \mid \text{RW}^+ \mid \text{void}$$

Below we describe the constraints that must be satisfied for each command to be executed successfully, and present the revised typing rule as follows:

`cp`: To successfully execute the command `cp` f_1 f_2 , the following constraints must be satisfied: *a*) The source file f_1 and the destination file f_2 must already exist in the system. *b*) The destination file f_2 must be either of type RW^- or WO^-. *c*) The source and destination files must exist in the system after executing the command. *d*) The type of the destination file f_2 must be changed to be the join of its type and the source type after executing the command. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau' \quad \tau' \sqsubseteq \text{WO}^-}{\Gamma \mid \Gamma'', f_1 : \tau, f_2 : \tau' \sqcup \tau \vdash \texttt{cp}\ f_1\ f_2 : \text{void}} (\texttt{cp})$$

`rd`: To successfully execute the command `rd` f, the following constraints must be satisfied: *a*) The file f must already exist in the system. *b*) The file f must be either of type RW^-, RW^+, or RO. *c*) The file f must not exist in the system after executing the command. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f : \tau \quad \tau \sqsubseteq \mathrm{RO}}{\Gamma \mid \Gamma' \vdash \texttt{rd}\ f : \mathrm{void}}\ (\texttt{rd})$$

`cat`: To successfully execute the command `cat` f_1 f_2 f_3 , the following constraints must be satisfied: *a*) The source files f_1 and f_2 and the destination file f_3 must already exist in the system. *b*) The source files f_1 and f_2 must be either of type RW^-,RW^+, WO^- or WO^+. *c*) The destination file f_3 must be either of type RW^- or WO^-. *d*) The source files f_1 and f_2 must not exist in the system after executing the command. *e*) The destination file f_3 must exist in the system after executing the command. *f*) The type of the destination file f_3 must be changed to be the join of its type and the types of the source files after executing the command. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau' \quad \Gamma'' \mid \Gamma''' \vdash f_3 : \tau'' \quad \tau, \tau' \sqsubseteq \mathrm{WO}^+ \wedge \tau'' \sqsubseteq \mathrm{WO}^-}{\Gamma \mid \Gamma''', f_3 : \tau \sqcup \tau' \sqcup \tau'' \vdash \texttt{cat}\ f_1\ f_2\ f_3 : \mathrm{void}}\ (\texttt{cat})$$

`mv`: To successfully execute the command `mv` f_1 f_2 , the following constraints must be satisfied: *a*) The source file f_1 and the destination file f_2 must already exist in the system. *b*) The destination file f_2 must be either of type RW^- or WO^-. *c*) The source file f_1 must not exist in the system after executing the command. *d*) The destination file f_2 must exist in the system after executing the command *e*) The type of the destination file f_2 must be changed to be the join of its type and the source type after executing the command. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau' \quad \tau' \sqsubseteq \mathrm{WO}^-}{\Gamma \mid \Gamma'', f_2 : \tau \sqcup \tau' \vdash \texttt{mv}\ f_1\ f_2 : \mathrm{void}}\ (\texttt{mv})$$

`append`: To successfully execute the command `append` f_1 f_2 f_3 , the following constraints must be satisfied: *a*) The source files f_1 and f_2 must already exist in the system. *b*) The destination file f_3 must not exist in the system. *c*) The source files f_1 and f_2 must be either of type RW^-,RW^+, WO^- or WO^+. *d*) The source files f_1 and f_2 must not exist in the system after executing the command. *e*) The destination file f_3 must exist in the system after executing the command. *f*) The type of the destination file f_3 must be the join of the types of the source files after executing the command. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau' \quad \tau, \tau' \sqsubseteq \mathrm{WO}^+}{\Gamma \mid \Gamma'', f_3 : \tau \sqcup \tau' \vdash \texttt{append}\ f_1\ f_2\ f_3 : \mathrm{void}}\ (\texttt{append})$$

The remaining typing rules are unchanged and depicted with the modified typing rules in Figure 6.2. It can be seen that the type system shown in Figure 6.2 is similar to the type system presented in Chapter 5 which focuses on regulating *copy* operations. The only difference is that the checking performed by the type system presented in this section is to control *read* and *write* operations and enforce their flow policies, whereas the checking performed by the previous type system is to control *copy* operations and enforce their flow policies. The typing algorithm for the previous system, therefore, can be used with a slight modification for checking the new types. We present a new version of the typing algorithm when we discuss combining the two type systems in the next section.

$$\frac{}{\Gamma, f : \tau \mid \Gamma \vdash f : \tau}\,(f) \qquad \frac{\Gamma \mid \Gamma' \vdash c : \text{void} \quad \Gamma' \mid \Gamma'' \vdash cs : \text{void}}{\Gamma \mid \Gamma'' \vdash c; cs : \text{void}}\,(cs)$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau' \quad \tau' \sqsubseteq \text{WO}^-}{\Gamma \mid \Gamma'', f_1 : \tau, f_2 : \tau' \sqcup \tau \vdash \texttt{cp}\ f_1\ f_2 : \text{void}}\,(\texttt{cp})$$

$$\frac{\Gamma \mid \Gamma' \vdash f : \tau}{\Gamma \mid \Gamma' \vdash \texttt{rm}\ f : \text{void}}\,(\texttt{rm}) \qquad \frac{}{\Gamma \mid \Gamma, f : t \vdash \texttt{mkf}\ f\ t : \text{void}}\,(\texttt{mkf}) \qquad \frac{\Gamma \mid \Gamma' \vdash f : \tau \quad \tau \sqsubseteq \text{RO}}{\Gamma \mid \Gamma' \vdash \texttt{rd}\ f : \text{void}}\,(\texttt{rd})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau' \quad \Gamma'' \mid \Gamma''' \vdash f_3 : \tau'' \quad \tau, \tau' \sqsubseteq \text{WO}^+ \wedge \tau'' \sqsubseteq \text{WO}^-}{\Gamma \mid \Gamma''', f_3 : \tau \sqcup \tau' \sqcup \tau'' \vdash \texttt{cat}\ f_1\ f_2\ f_3 : \text{void}}\,(\texttt{cat})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau' \quad \tau' \sqsubseteq \text{WO}^-}{\Gamma \mid \Gamma'', f_2 : \tau \sqcup \tau' \vdash \texttt{mv}\ f_1\ f_2 : \text{void}}\,(\texttt{mv})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau}{\Gamma \mid \Gamma', f_1 : \tau, f_2 : \tau \vdash \texttt{copy}\ f_1\ f_2 : \text{void}}\,(\texttt{copy})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau \quad \Gamma' \mid \Gamma'' \vdash f_2 : \tau' \quad \tau, \tau' \sqsubseteq \text{WO}^+}{\Gamma \mid \Gamma'', f_3 : \tau \sqcup \tau' \vdash \texttt{append}\ f_1\ f_2\ f_3 : \text{void}}\,(\texttt{append})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : \tau}{\Gamma \mid \Gamma', f_2 : \tau \vdash \texttt{move}\ f_1\ f_2 : \text{void}}\,(\texttt{move})$$

Figure 6.2: Typing rules for security access types

It can be seen from the typing rules in Figure 6.2 that if there is information flow from a source file to a destination file, then the security type of the source file is always transferred to the destination file and joined with its security type. This could occur even when the security type of the destination file is different from the source file. For example, if $T(f_1) = \text{RO}$ and $T(f_2) = \text{RW}^-$, then the command `cp` f_1 f_2 will type check and will result in f_2 changes its type to be $\text{RO} \sqcup \text{RW}^- = \text{RO}$. This might seem to be inconsistent with standard notions of integrity which are intended to preserve validity of an association between specific objects and their contents. For example, a file of type RW^- should only store information of type RW^-, and a file of type RO should only store information of type RO. However, our security types represent policies that dictate what can be done with files, and the standard notions of integrity contradict the policies imposed by our security types. For example, the security type RO dictates that a file associated with this type can only be read but not written into it. This is useful to avoid accidental overwriting of a file that is created to be read only. Therefore, if $T(f_1) = \text{RO}$ and $T(f_2) = \text{RO}$, then `cp` f_1 f_2 must not be allowed, even through the types of the source and destination files are the same. This is because the policy imposed by the security type RO will be violated if such command is executed. The only security types that allow a file to be overwritten are RW^- and WO^-. However, if we allow a file of type RW^- or WO^- to only store information of the same type, then copying a file of type RO, for example, will not be possible. This is because $\text{RO} \neq \text{RW}^- \vee \text{WO}^-$. We avoid this by allowing a file of type RW^- or WO^- to store information of any type. However, to preserve the integrity constraints of both the source and destination files, we require the destination file to change its type to be the join of the security type of the source file and the security type of the destination file.

In this way, the destination file will only store information of the same type or more restrictive type. In case of the information to be stored in a destination file is associated with less restrictive type than the type of the destination file, then the information will acquire the type of the destination file. For example, if $T(f_1) = \text{RW}^-$ and $T(f_2) = \text{WO}^-$, then the command `cp` f_1 f_2 will type check and will result in f_2 changes its type to be $\text{RW}^- \sqcup \text{WO}^- = \text{WO}^-$. On the other hand, if the information to be stored in a destination file is associated with more restrictive type than the type of the destination file, then the destination file will acquire the type of the information to be stored. For example, if $T(f_1) = \text{NRW}$ and $T(f_2) = \text{RW}^-$, then the command `cp` f_1 f_2 will type check and will result in f_2 changes its type to be $\text{NRW} \sqcup \text{RW}^- = \text{NRW}$.

Therefore, changing the type of the destination file to be the join of the type of the information to be stored and the type of the destination file plays an important rule in preserving the integrity of files. This is because the integrity constraints of the information to be stored and the integrity constraints of the destination file can only change to constraints that is at least as restrictive as both of them.

6.3 Security copy and access types

In this section we combine the two systems introduced so far to enforce their policies in one system. In the previous systems we assumed each file is associated with a type τ where $\tau \in \{\mathrm{UC}, \mathrm{LC}^n, \mathrm{NC}\} \vee \{\mathrm{NRW}, \mathrm{RO}, \mathrm{WO}^-, \mathrm{WO}^+, \mathrm{RW}^-, \mathrm{RW}^+\}$. The type system in Chapter 5 controls the access to *copy* operations and the flow caused by all operations including *copy*; while the type system in the previous section controls the access to *read* and *write* operations and the flow caused by all operations. In the system we introduce in this section we need to control the access to *copy*, *read*, and *write* operations as well as the flow caused by them. To achieve this we need to combine both lattices of the security types such that each file should be associated with two types. One type regulates *copy* operations and the other type regulates *read* and *write* operations. Therefore, we assume that each file in the system is associated with an ordered pair type of the form $f : (\alpha, \beta)$ where α is a security copy type and β is a security access type. This leads to the following definition of a pair type:

Definition 6.3.1. $\forall f \in types,\ T(f) = (\alpha, \beta)$,

where $\alpha \in \{\mathrm{UC}, \mathrm{LC}^n, \mathrm{NC}\} \wedge \beta \in \{\mathrm{NRW}, \mathrm{RO}, \mathrm{WO}^-, \mathrm{WO}^+, \mathrm{RW}^-, \mathrm{RW}^+\}$.

Based on this definition, the types $(\mathrm{UC}, \mathrm{RO})$ and $(\mathrm{NC}, \mathrm{NRW})$ are allowed because $(\mathrm{UC}, \mathrm{RO}) \rightarrow \mathrm{UC} \in \{\mathrm{UC}, \mathrm{LC}^n, \mathrm{NC}\} \wedge \mathrm{RO} \in \{\mathrm{NRW}, \mathrm{RO}, \mathrm{WO}^-, \mathrm{WO}^+, \mathrm{RW}^-, \mathrm{RW}^+\}$, and $(\mathrm{NC}, \mathrm{NRW}) \rightarrow \mathrm{NC} \in \{\mathrm{UC}, \mathrm{LC}^n, \mathrm{NC}\} \wedge \mathrm{NRW} \in \{\mathrm{NRW}, \mathrm{RO}, \mathrm{WO}^-, \mathrm{WO}^+, \mathrm{RW}^-, \mathrm{RW}^+\}$. On the other hand, the types $(\mathrm{UC}, \mathrm{LC})$ and $(\mathrm{WO}, \mathrm{RO})$ are not allowed because $(\mathrm{UC}, \mathrm{LC}) \rightarrow \mathrm{UC} \in \{\mathrm{UC}, \mathrm{LC}^n, \mathrm{NC}\} \wedge \mathrm{LC} \notin \{\mathrm{NRW}, \mathrm{RO}, \mathrm{WO}^-, \mathrm{WO}^+, \mathrm{RW}^-, \mathrm{RW}^+\}$, and $(\mathrm{WO}, \mathrm{RO}) \rightarrow \mathrm{WO} \notin \{\mathrm{UC}, \mathrm{LC}^n, \mathrm{NC}\} \wedge \mathrm{RO} \in \{\mathrm{NRW}, \mathrm{RO}, \mathrm{WO}^-, \mathrm{WO}^+, \mathrm{RW}^-, \mathrm{RW}^+\}$. It should be noted that all possible pair types that we could instantiate of the form $f : (\alpha, \beta)$ are useful in particular scenarios. The type α controls access to *copy* operations and controls the flow of information caused by all operations including *copy*. The type β controls access to *read* and *write* operations and control the flow of information caused by all operations including *read* and *write*. Considering the two security types α and β as an ordered pair type, allows us to control the access to the three operations *copy*, *read*, and *write* and control the flow of information caused by them and all other operations. In the next section we present the language and the typing rules to enforce the policies of both systems described previously.

6.3.1 Language and typing rules

In this section we revise the language and typing rules. The only difference in the revised language is in the command `mkf` and the type τ which is now a pair type (α, β). The command `mkf` now takes a file f, a security copy type α, and a security access type β, and creates a file f of type (α, β). The syntax of the language is as follows.

$$
\begin{array}{lll}
\langle p \rangle & ::= & \langle cs \rangle \mid \langle f \rangle \\
\langle cs \rangle & ::= & \langle c \rangle \mid \langle c \rangle ; \langle cs \rangle \\
\langle c \rangle & ::= & \texttt{cp}\ \langle f \rangle\ \langle f \rangle \mid \texttt{rm}\ \langle f \rangle \mid \texttt{mkf}\ \langle f \rangle\ \langle \alpha \rangle\ \langle \beta \rangle \mid \texttt{rd}\ \langle f \rangle \mid \texttt{cat}\ \langle f \rangle\ \langle f \rangle\ \langle f \rangle \mid \texttt{mv}\ \langle f \rangle\ \langle f \rangle \\
 & \mid & \texttt{copy}\ \langle f \rangle\ \langle f \rangle \mid \texttt{append}\ \langle f \rangle\ \langle f \rangle\ \langle f \rangle \mid \texttt{move}\ \langle f \rangle\ \langle f \rangle \\
\langle t \rangle & ::= & (\langle \alpha \rangle, \langle \beta \rangle) \mid \text{void} \\
\langle \alpha \rangle & ::= & \text{NC} \mid \text{LC}^n \mid \text{UC} \\
\langle \beta \rangle & ::= & \text{NRW} \mid \text{RO} \mid \text{WO}^- \mid \text{WO}^+ \mid \text{RW}^- \mid \text{RW}^+
\end{array}
$$

Figure 6.3 shows the new type system that enforces both policies presented in each previous system. This type system will not only enforce the number of times a file can be read, but also enforce the different types of access of the policies identified in Chapter 4. For example, based on the security types of files, the type system enforces which file can or cannot be read or written into it. In the next section we revise the typing algorithm presented in the previous chapter to accommodate the new version of the typing rules.

6.3.2 Typing algorithm

In this section we present a type inference algorithm $\mathcal{T}$ for typing phrases according to the new type system depicted in Figure 6.3. It is similar to the typing algorithm presented in the previous chapter, however, with slight modification to accommodate the new checking in the revised type system. We define the following functions: $\mathit{check}(\tau, \tau')$ returns true if the types are compatible. $\mathit{less}(\tau, \tau')$ returns true if $\tau \sqsubseteq \tau'$. $\mathit{lub}(\tau, \dots, \tau_n)$ returns the least upper bound of all its parameters i.e. $\tau \sqcup \dots \sqcup \tau_n$. Note that if $\tau = (\tau_1, \tau_2)$ and $\tau' = (\tau_1', \tau_2')$, then $\mathit{lub}(\tau, \tau') = (\tau_1 \sqcup \tau_1', \tau_2 \sqcup \tau_2')$. $\pi_1(\tau)$ and $\pi_2(\tau)$ returns the security copy type and access type of τ, respectively. That is, if $\tau = (\tau_1, \tau_2)$, then $\pi_1(\tau) = \tau_1$ and $\pi_2(\tau) = \tau_2$. Finally, the functions *red* and *dst* are as defined before, however, for simplicity when applied to a pair type, they should be understood as applying to the security copy type of the pair only. For example, if $\tau = (\tau_1, \tau_2)$, then $\mathit{red}(\tau) = (\mathit{red}(\tau_1), \tau_2)$. Using these functions, we can now define the type inference algorithm $\mathcal{T}$ as follows:

The Type Reconstruction Algorithm $\mathcal{T}$:

$$\frac{}{\Gamma, f : (\alpha, \beta) \mid \Gamma \vdash f : (\alpha, \beta)}\,(f) \qquad \frac{\Gamma \mid \Gamma' \vdash c : \mathrm{void} \quad \Gamma' \mid \Gamma'' \vdash cs : \mathrm{void}}{\Gamma \mid \Gamma'' \vdash c; cs : \mathrm{void}}\,(cs)$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : (\alpha, \beta) \quad \alpha \sqsubseteq \mathrm{LC}^{n>0} \quad \Gamma' \mid \Gamma'' \vdash f_2 : (\alpha', \beta') \quad \beta' \sqsubseteq \mathrm{WO}^-}{\Gamma \mid \Gamma'', f_1 : (\mathit{red}(\alpha), \beta), f_2 : (\mathit{dst}(\alpha) \sqcup \alpha', \beta \sqcup \beta') \vdash \texttt{cp}\ f_1\ f_2 : \mathrm{void}}\,(\texttt{cp})$$

$$\frac{\Gamma \mid \Gamma' \vdash f : (\alpha, \beta)}{\Gamma \mid \Gamma' \vdash \texttt{rm}\ f : \mathrm{void}}\,(\texttt{rm}) \quad \frac{}{\Gamma \mid \Gamma, f : (\alpha, \beta) \vdash \texttt{mkf}\ f\ \alpha\ \beta : \mathrm{void}}\,(\texttt{mkf}) \quad \frac{\Gamma \mid \Gamma' \vdash f : (\alpha, \beta) \quad \beta \sqsubseteq \mathrm{RO}}{\Gamma \mid \Gamma' \vdash \texttt{rd}\ f : \mathrm{void}}\,(\texttt{rd})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : (\alpha, \beta) \quad \Gamma' \mid \Gamma'' \vdash f_2 : (\alpha', \beta') \quad \Gamma'' \mid \Gamma''' \vdash f_3 : (\alpha'', \beta'') \quad \beta, \beta' \sqsubseteq \mathrm{WO}^+ \wedge \beta'' \sqsubseteq \mathrm{WO}^-}{\Gamma \mid \Gamma''', f_3 : (\alpha \sqcup \alpha' \sqcup \alpha'', \beta \sqcup \beta' \sqcup \beta'') \vdash \texttt{cat}\ f_1\ f_2\ f_3 : \mathrm{void}}\,(\texttt{cat})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : (\alpha, \beta) \quad \Gamma' \mid \Gamma'' \vdash f_2 : (\alpha', \beta') \quad \beta' \sqsubseteq \mathrm{WO}^-}{\Gamma \mid \Gamma'', f_2 : (\alpha \sqcup \alpha', \beta \sqcup \beta') \vdash \texttt{mv}\ f_1\ f_2 : \mathrm{void}}\,(\texttt{mv})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : (\alpha, \beta) \quad \alpha \sqsubseteq \mathrm{LC}^{n>0}}{\Gamma \mid \Gamma', f_1 : (\mathit{red}(\alpha), \beta), f_2 : (\mathit{dst}(\alpha), \beta) \vdash \texttt{copy}\ f_1\ f_2 : \mathrm{void}}\,(\texttt{copy})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : (\alpha, \beta) \quad \Gamma' \mid \Gamma'' \vdash f_2 : (\alpha', \beta') \quad \beta, \beta' \sqsubseteq \mathrm{WO}^+}{\Gamma \mid \Gamma'', f_3 : (\alpha \sqcup \alpha', \beta \sqcup \beta') \vdash \texttt{append}\ f_1\ f_2\ f_3 : \mathrm{void}}\,(\texttt{append})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : (\alpha, \beta)}{\Gamma \mid \Gamma', f_2 : (\alpha, \beta) \vdash \texttt{move}\ f_1\ f_2 : \mathrm{void}}\,(\texttt{move})$$

Figure 6.3: Typing rules for security copy and access types

$$\mathcal{T}(A, e) = (\tau, A')$$

where:

1. If e is the filename f, and $f : \alpha \in A$ then $\tau = \alpha$, $A' = A \setminus \{f : \alpha\}$.

2. If e is a sequence of commands, $c; cs$ let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, c) \\ \mathit{check}(\alpha, \mathrm{void}) & & \\ (\beta, A_2) & = & \mathcal{T}(A_1, cs) \\ \mathit{check}(\beta, \mathrm{void}) & & \end{array}$$

then $\tau = \mathrm{void}, A' = A_2$.

3. If e is the cp command, `cp` f_1 f_2 let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, f_1) \\ less(\pi_1(\alpha), \mathrm{LC}^{n>0}) & & \\ (\beta, A_2) & = & \mathcal{T}(A_1, f_2) \\ less(\pi_2(\beta), \mathrm{WO}^-) & & \end{array}$$

   then if $f_1, f_2 \notin A_2$, then $\tau = \text{void}, A' = A_2 \cup \{f_1 : \mathit{red}(\alpha), f_2 : \mathit{lub}(\mathit{dst}(\alpha), \beta)\}$.

4. If e is the rm command, `rm` f let

$$(\alpha, A_1) \quad = \quad \mathcal{T}(A, f)$$

   then $\tau = \text{void}, A' = A_1$.

5. If e is the mkf command, `mkf` f α, then if $f \notin A$, then $\tau = \text{void}, A' = A \cup \{f : \alpha\}$.

6. If e is the rd command, `rd` f let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, f) \\ less(\pi_2(\alpha), \mathrm{RO}) & & \end{array}$$

   then $\tau = \text{void}, A' = A_1$.

7. If e is the cat command, `cat` f_1 f_2 f_3 let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, f_1) \\ (\beta, A_2) & = & \mathcal{T}(A_1, f_2) \\ (\delta, A_3) & = & \mathcal{T}(A_2, f_3) \\ less(\pi_2(\alpha), \mathrm{WO}^+) & & \\ less(\pi_2(\beta), \mathrm{WO}^+) & & \\ less(\pi_2(\delta), \mathrm{WO}^-) & & \end{array}$$

   then if $f_3 \notin A_3$, then $\tau = \text{void}, A' = A_3 \cup \{f_3 : \mathit{lub}(\alpha, \beta, \delta)\}$.

8. If e is the mv command, `mv` f_1 f_2 let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, f_1) \\ (\beta, A_2) & = & \mathcal{T}(A_1, f_2) \\ less(\pi_2(\beta), \mathrm{WO}^-) & & \end{array}$$

   then if $f_2 \notin A_2$, then $\tau = \text{void}, A' = A_2 \cup \{f_2 : \mathit{lub}(\alpha, \beta)\}$.

9. If e is the copy command, `copy` f_1 f_2 let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, f_1) \\ less(\pi_1(\alpha), \mathrm{LC}^{n>0}) & & \end{array}$$

then if $f_2 \notin A$, then $\tau = \mathrm{void}, A' = A_1 \cup \{f_1 : red(\alpha), f_2 : dst(\alpha)\}$.

10. If e is the append command, `append` f_1 f_2 f_3 let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, f_1) \\ (\beta, A_2) & = & \mathcal{T}(A_1, f_2) \\ less(\pi_2(\alpha), \mathrm{WO}^+) & & \\ less(\pi_2(\beta), \mathrm{WO}^+) & & \end{array}$$

then if $f_3 \notin A$, then $\tau = \mathrm{void}, A' = A_2 \cup \{f_3 : lub(\alpha, \beta)\}$.

11. If e is the move command, `move` f_1 f_2 let

$$(\alpha, A_1) = \mathcal{T}(A, f_1)$$

then if $f_2 \notin A$, then $\tau = \mathrm{void}, A' = A_1 \cup \{f_2 : \alpha\}$.

6.4 Ownership and authorisation

In the previous section we presented a type system to control the access to *copy*, *read*, and *write* operations and the flow of information that is caused by all operations. The type system in the previous section is missing two important aspects that are related to each other. These two aspects are *ownership* and *authorisation.* The former indicates the owners of files while the latter indicates the users authorised by the owners to perform particular operations on their files. Although the security copy and access types presented in the previous sections offer some sort of authorisation by controlling which operations can be performed on which types of files, they are not concerned with who can perform these operations. For example, consider the case where *Alice* is a user and has two private files f_1 and f_2. *Alice* wants f_1 to be read only by *Bob* and wants f_2 to be read only by *Carol.* To prevent these two files from any modification, *Alice* can give the following type to her files $(\mathrm{NC}, \mathrm{RO})$, so that only *read* operations can be performed on them. However, although *Alice* is not willing to allow *Carol* to read f_1, *Carol* can read f_1 by issuing the command `rd` f_1 since the type of the f_1 allows such operation to be performed. Therefore, *Alice* will not be able to specify these policies and the two files can be read by any user.

The security copy types and security access types can control which operations can be performed on which types of files, but not which users can perform which operations on which types of files. For *Alice* to specify her policy in the above example, the ownership and authorisation aspects should be incorporated into the type, such that an owner of a file can specify which user is allowed to perform the operations allowed by the type of the file. In this section we augment the security copy types and the security access types with ownership and authorisation aspects. The users authorised to perform the allowed operations on files must be specified by the owners of the files. Files are owned by users who created them. To control the allowed operations so that only the authorised users can perform them, the type of the file must contain the owners and the authorised users of that file. Our approach to incorporate the ownership and the authorisation information into the types of the files is inspired by earlier work of Myers and Liskov [75, 76, 74, 77] who develop a decentralised model for information flow known as *the decentralised label model* (DLM).

6.4.1 Label structure

Our security types of files which represent the security policies will be expressed in a label. Similar to DLM, a label consists of one or more components; each component representing a file type which is a policy. However, the structure and the interpretation of labels are different from the conventional structure and interpretation of DLM. A label with one policy has the following form.

$$l_1 = \{\alpha,\ \beta,\ \delta\}$$

where α and β are a security copy type and a security access type, respectively, which are discussed in the previous sections. δ represents the ownership and authorisation information for this policy. It has exactly the same structure as the entire label of DLM, however, with different interpretation. δ has the following form.

$$o : u_1, u_2, u_3$$

where o is the owner and u_1, u_2, u_3 are the authorised users specified by the owner. Thus, the label l_1 is expressed as follows.

$$l_1 = \{\alpha,\ \beta,\ o : u_1, u_2, u_3\}$$

the interpretation of such label is that the file owner o authorises the users u_1, u_2, u_3 to perform operations on the file that are allowed by the security types α and β. Therefore, users will not be able to perform any operations arbitrarily, rather the operations to be performed must be authorised by the security types and the information flow caused by them must not violate the lattice structure of either of the security types. In other words, δ controls which users are allowed to perform operations on the file, while α and β control which operations can be performed on the file and control the flow of information that results from these operations. A concrete example for a label with one policy is the following.

$$l_2 = \{\text{UC},\ \text{RO},\ \mathit{Alice} : \mathit{Bob}, \mathit{Carol}, \mathit{Dave}\}$$

the single policy in label l_2 states that the owner *Alice* allows $\mathit{Bob}, \mathit{Carol}$, and *Dave* to only copy and read the file associated with this label. If there is another user *Eve* who issues *read* or *copy* operations to copy or read the file, these will fail. This is because *Eve* is not one of the authorised users who are specified by the owner *Alice* to perform these operations.

The single policy label that is described above is assumed to be attached to a file at the time of creation. That is the single policy label will be attached to a file when the command `mkf` is issued by a user. Therefore, each file in the system is assumed to be attached with a label. The owner of such a policy will be the user who issued the command `mkf`. However, a label might contain multiple policies as a result of information flow caused by any operation. For example, if $f_1 \wedge f_2 \in \mathit{Types}$, which means both files are associated with a label, then flow of information from f_1 to f_2, must result in f_2 changing its label to enforce all the policies of its label and the label of f_1. To formally define the components and properties of our label we use the following notations. $o\delta(J)$ denotes the owner of the policy J. $u\delta(J)$ denotes the set of authorised users of the policy J including the owner. $\alpha(J)$ denotes the security copy type of the policy J. $\beta(J)$ denotes the security access type of the policy J. Hence, if we assume J is the following single policy label:

$$l_3 = \{\text{NC},\ \text{WO}^-,\ \mathit{Alice} : \mathit{Bob}, \mathit{Carol}\}$$

then, $o\delta(J) = \mathit{Alice}$, $u\delta(J) = \mathit{Alice}, \mathit{Bob}, \mathit{Carol}$, $\alpha(J) = \text{NC}$, $\beta(J) = \text{WO}$. However, as a result of information flow between two files, two single label policies are combined and lead to a label that consists of more than one policy. For simplicity we assume that label l consists of two polices J and K, however, this can be applied to any number of policies. Below we define the properties of our label more generally to encompass a multiple policy label. We assume the following label as running example for the definitions below:

$$l_4 = \{\text{LC}^3, \text{RO},\ \mathit{Alice} : \mathit{Bob};\ \text{UC}, \text{WO}^-,\ \mathit{Bob} : \mathit{Carol}\}$$

Definition 6.4.1. The set of owners of a label l denoted as $o\delta(l)$ is the union of all owners of each policy in the label l.

$$o\delta(l) = o\delta(K) \sqcup o\delta(J)$$

The owner set of a label is the owner of each policy in the label. For example $o\delta(l_4) = o\delta(Alice) \sqcup o\delta(Bob) = (Alice, Bob)$. Therefore, only *Alice* and *Bob* are the owners of label l_4.

Definition 6.4.2. The set of effective authorised users of a label l denoted as $u\delta(l)$ is the intersection of all authorised users of each policy in the label l.

$$u\delta(l) = u\delta(K) \sqcap u\delta(J)$$

The set of effective authorised users are those who are agreed by each policy owner as authorised users. That is, each policy owner must specify these users as authorised users in his policy. For example, $u\delta(l_4) = u\delta(Alice, Bob) \sqcap u\delta(Bob, Carol) = (Bob)$. Therefore, only *Bob* is the effective authorised user in label l_4 as both policies agree on this.

Definition 6.4.3. The effective security copy type of a label l denoted as $\alpha(l)$ is the join of all security copy types of each policy in the label l.

$$\alpha(l) = \alpha(K) \sqcup \alpha(J)$$

The effective security copy type is the type that is at least as restrictive as the security copy type of each policy in the label l. That is, the least upper bound or join of the security copy types of every policy in the label l. For example, $\alpha(l_4) = \alpha(\text{LC}^3) \sqcup \alpha(\text{UC}) = (\text{LC}^3)$. Therefore, LC^3 will be the security copy type of label l_4.

Definition 6.4.4. The effective security access type of a label l denoted as $\beta(l)$ is the join of all security access types of each policy in the label l.

$$\beta(l) = \beta(K) \sqcup \beta(J)$$

The effective security access type is the type that is at least as restrictive as the security access type of each policy in the label l. That is, the least upper bound or join of the security access types of every policy in the label l. For example, $\beta(l_4) = \beta(\text{RO}) \sqcup \beta(\text{WO}^-) = (\text{NRW})$. Therefore, NRW will be the security access type in label l_4.

Definition 6.4.5. The effective policy of a label l denoted as $ep(l)$ is the intersection of all authorised users in each policy of l, the join of all security copy types in each policy of l, and the join of all security access types in each policy of l.

$$ep(l) = \alpha(l) \wedge \beta(l) \wedge u\delta(l)$$

The effective policy is the policy that is derived from a label of multiple policies. This policy is the only one to consider when evaluating a multiple policy label. For example, $ep(l_4) = \alpha(l_4) \wedge \beta(l_4) \wedge u\delta(l_4) = (\mathrm{LC}^3; \mathrm{NRW}; Bob)$.

Definition 6.4.6. The union of two labels l and l', written as $l \sqcup l'$, is the set of all policies that exist in both l and l'.

$$l \sqcup l' = l \cup l'$$

For example, if $l = \{\mathrm{LC}^2, \mathrm{RW}^+,\ Alice : Bob, Carol; \mathrm{UC}, \mathrm{RO},\ Bob : Carol\}$ and $l' = \{\mathrm{NC}, \mathrm{RO},\ Bob : Carol\}$, then $l \sqcup l' = \{\mathrm{LC}^2, \mathrm{RW}^+,\ Alice : Bob, Carol; \mathrm{UC}, \mathrm{RO},\ Bob : Carol; \mathrm{NC}, \mathrm{RO},\ Bob : Carol\}$.

Definition 6.4.7. The effective authorised users of a label l' is a subset of the effective authorised users of a label l, written as $u\delta(l) \sqsubseteq u\delta(l')$, if and only if every effective authorised user in l' is also an effective authorised user in l.

$$u\delta(l) \sqsubseteq u\delta(l') = u\delta(l') \subseteq u\delta(l)$$

For example, if $l = \{\mathrm{LC}^2, \mathrm{RW}^+,\ Alice : Bob, Carol\}$ and $l' = \{\mathrm{NC}, \mathrm{RO},\ Alice : Bob\}$, then $(Alice, Bob, Caro) \sqsubseteq (Alice, Bob)$.

Definition 6.4.8. The function *red* if applied to label l, written as $red(l)$, changes the security copy type α in each policy in l to be $red(\alpha(l))$.

$$red(l) = \forall J \in l, \alpha(J) = red(\alpha(l))$$

For example, $red(l_4) = \{\mathrm{LC}^2, \mathrm{RO},\ Alice : Bob;\ \mathrm{LC}^2, \mathrm{WO}^-,\ Bob : Carol\}$.

Definition 6.4.9. The function *dst* if applied to label l, written as $dst(l)$, changes the security copy type α in each policy in l to be $dst(\alpha(l))$.

$$dst(l) = \forall J \in l, \alpha(J) = dst(\alpha(l))$$

For example, $dst(l_4) = \{\mathrm{NC}, \mathrm{RO},\ Alice : Bob;\ \mathrm{NC}, \mathrm{WO}^-,\ Bob : Carol\}$.

Based on these definitions, previous policies to control *copy*, *read*, and *write* operations can be stated straightforwardly by only considering the effective policies of labels. For example, a file f associated with label l can be read if $\beta(l) \in \{\mathrm{RW}^-, \mathrm{RW}^+, \mathrm{RO}\}$, overwritten if $\beta(l) \in \{\mathrm{RW}^-, \mathrm{WO}^-\}$, appended to it if $\beta(l) \in \{\mathrm{RW}^-, \mathrm{RW}^+\mathrm{WO}^-, \mathrm{WO}^+\}$, or copied if $\alpha(l) \in \{\mathrm{UC}, \mathrm{LC}^{n>0}\}$. Our aim for introducing the label structure in this section is not to change previous policies, but rather to restrict who can exercise them. Therefore, we extend the previous policies to control *copy*, *read*, and *write* operations with the following: *a*) Operations can be performed if and only if the user issuing them is one of the effective authorised users of all the label associated with the files the operations applied to. *b*) Flows of information from a source file to a destination file caused by any operation is allowed if and only if the effective authorised users of the label associated with the destination file is a subset of the effective authorised users of the label associated with the source file. *c*) Flows of information from a source file to a destination file caused by any operations must change the label associated with the destination file to be the union of its label and the label associated with the source file. *d*) A newly created file should be assigned the user who created it to be the owner and the only authorised user of the policy associated with the created file. *e*) Files can be deleted or renamed if and only if the user issuing these operations is one of the owners of the label associated with the files the operations applied to.

In the next section we present the revised language and typing rules to enforce the policy described above

6.4.2 Language and typing rules

The syntax of the language is given by the following grammar:

$$
\begin{array}{lcl}
\langle p\rangle & ::= & \langle u\rangle.\langle cs\rangle \mid \langle f\rangle \\
\langle cs\rangle & ::= & \langle c\rangle \mid \langle c\rangle;\langle cs\rangle \\
\langle c\rangle & ::= & \texttt{cp}\ \langle f\rangle\ \langle f\rangle \mid \texttt{rm}\ \langle f\rangle \mid \texttt{mkf}\ \langle\alpha\rangle\ \langle\beta\rangle \mid \texttt{rd}\ \langle f\rangle \mid \texttt{cat}\ \langle f\rangle\ \langle f\rangle\ \langle f\rangle \mid \texttt{mv}\ \langle f\rangle\ \langle f\rangle \\
 & \mid & \texttt{copy}\ \langle f\rangle\ \langle f\rangle \mid \texttt{append}\ \langle f\rangle\ \langle f\rangle\ \langle f\rangle \mid \texttt{move}\ \langle f\rangle\ \langle f\rangle \\
\langle\tau\rangle & ::= & \langle l\rangle \mid \text{void} \\
\langle l\rangle & ::= & \langle pl\rangle \mid \langle pl\rangle; l \\
\langle pl\rangle & ::= & \langle\alpha\rangle, \langle\beta\rangle, \langle\delta\rangle \\
\langle\alpha\rangle & ::= & \text{NC} \mid \text{LC}^n \mid \text{UC} \\
\langle\beta\rangle & ::= & \text{NRW} \mid \text{RO} \mid \text{WO}^- \mid \text{WO}^+ \mid \text{RW}^- \mid \text{RW}^+ \\
\langle\delta\rangle & ::= & \langle u\rangle : \langle u\rangle, \langle u\rangle, \langle u\rangle, \ldots
\end{array}
$$

It can be seen that commands are unchanged and the same as described in the previous section. The only difference is in the $\langle u\rangle$ which ranges over the set of users' names for a given file system and the structure of file types which are represented as labels $\langle l\rangle$. A label can be of a single policy *pl*, or multiple policies pl_1, pl_2, pl_3, pl_n. Each policy consists of a security copy type α, a security access type β, and an ownership and authorisation type

δ. The new typing judgements have the following form:

$$\Gamma \mid \Gamma' \vdash u.cs : \tau \qquad \Gamma \mid \Gamma' \vdash f : \tau$$

where Γ is a set of files with labels of the form $f : l$. For example, $\Gamma = \{f_1 : l_1, f_2 : l_2, f_3 : l_3, \ldots, f_n : l_n\}$. The judgment $\Gamma \mid \Gamma' \vdash u.cs : \tau$ means that typing a single or a sequence of commands of type τ in the context Γ by user u, will change the context to Γ'. Similarly, the judgement $\Gamma \mid \Gamma' \vdash f : \tau$ means that typing a file in the context Γ will change the context to Γ'. The typing rules for a file name and a sequence of commands are the same as follows:

$$\frac{}{\Gamma, f : l \mid \Gamma \vdash f : l}\,(f) \qquad \frac{\Gamma \mid \Gamma' \vdash u.c : \text{void} \qquad \Gamma' \mid \Gamma'' \vdash u.cs : \text{void}}{\Gamma \mid \Gamma'' \vdash u.c; u.cs : \text{void}}\,(cs)$$

The remaining revised typing rules are given together with a description of the additional constraints for each of the commands below:

`cp` **command:** To successfully execute the command `cp` f_1 f_2, the following constraints must be satisfied: *a*) The source file f_1 and the destination file f_2 must already exist in the system. *b*) The effective security copy type of the label associated with the file f_1 must be either of type UC or $\text{LC}^{n>0}$. *c*) The user who issues the command must be one of the effective authorised users of the labels associated with f_1 and f_2. *d*) The effective authorised users of the label associated with the file f_2 must be a subset of the effective authorised users of the label associated with the file f_1. *e*) The effective security access type of the label associated with the file f_2 must be either of type RW^- or WO^-. *f*) The source and destination files must exist after executing the command. *g*) After executing the command, the security copy type of each policy in the label associated with the file f_1 must be changed to be the same as the type resulted from applying the function *red* to the the effective security copy type of the label associated with the file f_1. *h*) After executing the command, the label associated with the file f_2 must be changed to be the union of its label and the label of the file f_1, where the security copy type of each policy in the label associated with the file f_1 must be changed to be the same as the type resulted from applying the function *dst* to its effective security copy type before taking the union. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : l \quad \alpha(l) \sqsubseteq \text{LC}^{n>0} \quad \Gamma' \mid \Gamma'' \vdash f_2 : l' \quad \begin{array}{c} u \in u\delta(l) \wedge u\delta(l') \\ u\delta(l) \sqsubseteq u\delta(l') \end{array} \quad \beta(l') \sqsubseteq \text{WO}^-}{\Gamma \mid \Gamma'', f_1 : \mathit{red}(l), f_2 : l' \sqcup \mathit{dst}(l) \vdash u.\texttt{cp}\ f_1\ f_2 : \text{void}}\,(\texttt{cp})$$

`rm` **command:** To successfully execute the command `rm` f, the following constraints must be satisfied: *a*) The file f must already exist in the system. *b*) The user who issues the command must be one of the owners of the label associated with the file f. *c*) The file f must not exist in the system after executing the command. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad \mathtt{u} \in o\delta(l)}{\Gamma \mid \Gamma' \vdash u.\mathtt{rm}\ f : \text{void}}\ (\mathtt{rm})$$

`mkf` f α β **command:** To successfully execute the command `mkf` $f\ \alpha\ \beta$, the following constraints must be satisfied: *a*) The file f must not exist in the system. *b*) The file f must exist in the system after executing the command. *c*) The file f must be associated with a label of the form (α, β, δ), where α and β must be the security copy type and the security access type, respectively, that are specified at the time the command `mkf` is issued. Whereas δ is the ownership and the authorisation component of the label that must be of the form $u : u$, where u must be the user who issued the command. In other words, the command `mkf f` α β will automatically set the user who executed it to be the owner and the only authorised user. This leads to the following typing rule:

$$\frac{}{\Gamma \mid \Gamma, f : (\alpha, \beta, u : u) \vdash u.\mathtt{mkf}\ f\ \alpha\ \beta : \text{void}}\ (\mathtt{mkf})$$

`rd` **command:** To successfully execute the command `rd` f, the following constraints must be satisfied: *a*) The file f must already exist in the system. *b*) The user who issues the command must be one of the effective authorised users of the label associated with the file f. *c*) The file f must not exist in the system after executing the command. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad \mathtt{u} \in u\delta(l) \quad \beta(l) \sqsubseteq \text{RO}}{\Gamma \mid \Gamma' \vdash u.\mathtt{rd}\ f : \text{void}}\ (\mathtt{rd})$$

`cat` **command:** To successfully execute the command `cat` $f_1\ f_2\ f_3$, the following constraints must be satisfied: *a*) The source files f_1 and f_2, and the destination file f_3 must already exist in the system. *b*) The user who issues the command must be one of the effective authorised users of the labels associated with the files f_1, f_2, and f_3. *c*) The effective authorised users of the label associated with the file f_3 must be a subset of the effective authorised users of both labels associated with the files f_1 and f_2. *d*) The effective security access type of the labels associated with the source files f_1 and f_2 must be either of type RW$^-$,RW$^+$, WO$^-$ or WO$^+$. *e*) The security access type of the label associated with the destination file f_3 must either be of type RW$^-$ or WO$^-$. *f*) The source files f_1

and f_2 must not exist in the system after executing the command. *g*) The destination file f_3 must exist in the system after executing the command, and its label must be changed to be the union of its label and the labels of the source files f_1 and f_2. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : l \quad \Gamma' \mid \Gamma'' \vdash f_2 : l' \quad \Gamma'' \mid \Gamma''' \vdash f_3 : l'' \quad \begin{array}{c} u \in u\delta(l) \wedge u\delta(l') \wedge u\delta(l'') \\ u\delta(l) \wedge u\delta(l') \sqsubseteq u\delta(l'') \end{array} \quad \begin{array}{c} \beta(l) \wedge \beta(l') \sqsubseteq \mathrm{WO}^{+} \\ \beta(l'') \sqsubseteq \mathrm{WO}^{-} \end{array}}{\Gamma \mid \Gamma''', f_3 : l \sqcup l' \sqcup l'' \vdash u.\texttt{cat}\ f_1\ f_2\ f_3 : \mathrm{void}} \ (\texttt{cat})$$

`mv` **command:** To successfully execute the command `mv` f_1 f_2, the following constraints must be satisfied: *a*) The source file f_1 and the destination file f_2 must already exist in the system. *b*) The user who issues the command must be one of the owners of the label associated with the source file f_1, and one of the authorised users of the label associated with the destination file f_2. *c*) The effective authorised users of the label associated with the file f_2 must be a subset of the effective authorised users of the label associated with the file f_1. *d*) The effective security access type of the label associated with the file f_2 must be either of type RW^{-} or WO^{-}. *e*) The source file must not exist after executing the command. *f*) The destination file must exist after executing the command and its label must be changed to be the union of its label and the label associated with the file f_1, after executing the command. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : l \quad \Gamma' \mid \Gamma'' \vdash f_2 : l' \quad u \in o\delta(l) \wedge u\delta(l') \quad u\delta(l) \sqsubseteq u\delta(l') \quad \beta(l') \sqsubseteq \mathrm{WO}^{-}}{\Gamma \mid \Gamma'', f_2 : l' \sqcup l \vdash u.\texttt{mv}\ f_1\ f_2 : \mathrm{void}} \ (\texttt{mv})$$

`copy` **command:** To successfully execute the command `copy` f_1 f_2, the following constraints must be satisfied: *a*) The source file f_1 must already exist in the system. *b*) The destination file f_2 must not exist in the system. *c*) The effective security copy type of the label associated with the file f_1 must be either of type UC or $\mathrm{LC}^{n>0}$. *d*) The user who issues the command must be one of the effective authorised users of the label associated with f_1. *e*) The source and destination files f_1 and f_2 must exist after executing the command. *f*) After executing the command, the security copy type of each policy in the label associated with the file f_1 must be changed to be the same as the type resulting from applying the function *red* to the effective security copy type of the label associated with the file f_1. *g*) After executing the command, the file f_2 must be assigned the union of the label associated with the file f_1 applied to it *dst* function, and a label where its security copy type is the effective security copy type of the label associated with the file f_1 applied to it *dst* function, and its security access type is the effective security access type of the label associated with the file f_1, and the owner and the authorised users are the only user who issues the command. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : l \quad \alpha(l) \sqsubseteq \mathrm{LC}^{n>0} \quad u \in u\delta(l)}{\Gamma \mid \Gamma', f_1 : \mathit{red}(l), f_2 : \mathit{dst}(\alpha(l), \beta(l), u : u) \sqcup \mathit{dst}(l) \vdash u.\texttt{copy}\ f_1\ f_2 : \mathrm{void}}\ (\texttt{copy})$$

`append` **command:** To successfully execute the command `append` f_1 f_2 f_3, the following constraints must be satisfied: *a*) The source files f_1 and f_2 must already exist in the system. *b*) The destination file f_3 must not exist in the system. *c*) The user who issues the command must be one of the effective authorised users of the labels associated with the files f_1, and f_2. *d*) The effective security access type of the labels associated with the source files f_1 and f_2 must be either of type RW^-,RW^+, WO^- or WO^+. *e*) The source files f_1 and f_2 must not exist in the system after executing the command. *f*) The destination file f_3 must exist in the system after executing the command, and must be assigned the union of the labels of the source files f_1 and f_2, and a label where its security copy type is the join of the effective security copy type of the label associated with the files f_1 and f_2, its security access type is the join of the effective security access type of the label associated with the files f_1 and f_2, and the owner and the authorised users are the only user who issues the command. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : l \quad \Gamma' \mid \Gamma'' \vdash f_2 : l' \quad u \in u\delta(l) \wedge u\delta(l') \quad \beta(l) \wedge \beta(l') \sqsubseteq \mathrm{WO}^+}{\Gamma \mid \Gamma'', f_3 : (\alpha(l) \sqcup \alpha(l'), \beta(l) \sqcup \beta(l), u : u) \sqcup l \sqcup l' \vdash u.\texttt{append}\ f_1\ f_2\ f_3 : \mathrm{void}}\ (\texttt{append})$$

`move` **command:** To successfully execute the command `move` f_1 f_2, the following constraints must be satisfied: *a*) The source files f_1 must already exist in the system. *b*) The destination file f_2 must not exist in the system. *c*) The user who issues the command must be one of the owners of the label associated with the source file f_1. *d*) The source file must not exist after executing the command. *e*) The destination file must exist after executing the command and must be assigned the label associated with the file f_1. This leads to the following typing rule.

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f_2 : l \vdash u.\texttt{move}\ f_1\ f_2 : \mathrm{void}}\ (\texttt{move})$$

The typing rules are shown together in Figure 6.4. In the next section we introduce new commands along with their typing rules for manipulating file policies.

$$\frac{}{\Gamma, f : l \mid \Gamma \vdash f : l}\,(f) \qquad \frac{\Gamma \mid \Gamma' \vdash u.c : \text{void} \qquad \Gamma' \mid \Gamma'' \vdash u.cs : \text{void}}{\Gamma \mid \Gamma'' \vdash u.c; u.cs : \text{void}}\,(cs)$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : l \quad \alpha(l) \sqsubseteq \mathrm{LC}^{n>0} \quad \Gamma' \mid \Gamma'' \vdash f_2 : l' \qquad \begin{array}{c} u \in u\delta(l) \wedge u\delta(l') \\ u\delta(l) \sqsubseteq u\delta(l') \end{array} \qquad \beta(l') \sqsubseteq \mathrm{WO}^-}{\Gamma \mid \Gamma'', f_1 : \mathit{red}(l), f_2 : l' \sqcup \mathit{dst}(l) \vdash u.\texttt{cp}\ f_1\ f_2 : \text{void}}\,(\texttt{cp})$$

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad \mathtt{u} \in o\delta(l)}{\Gamma \mid \Gamma' \vdash u.\texttt{rm}\ f : \text{void}}\,(\texttt{rm}) \qquad \frac{}{\Gamma \mid \Gamma, f : (\alpha, \beta, u : u) \vdash u.\texttt{mkf}\ f\ \alpha\ \beta : \text{void}}\,(\texttt{mkf})$$

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad \mathtt{u} \in u\delta(l) \quad \beta(l) \sqsubseteq \mathrm{RO}}{\Gamma \mid \Gamma' \vdash u.\texttt{rd}\ f : \text{void}}\,(\texttt{rd})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : l \quad \Gamma' \mid \Gamma'' \vdash f_2 : l' \quad \Gamma'' \mid \Gamma''' \vdash f_3 : l'' \quad \begin{array}{c} u \in u\delta(l) \wedge u\delta(l') \wedge u\delta(l'') \\ u\delta(l) \wedge u\delta(l') \sqsubseteq u\delta(l'') \end{array} \quad \begin{array}{c} \beta(l) \wedge \beta(l') \sqsubseteq \mathrm{WO}^+ \\ \beta(l'') \sqsubseteq \mathrm{WO}^- \end{array}}{\Gamma \mid \Gamma''', f_3 : l \sqcup l' \sqcup l'' \vdash u.\texttt{cat}\ f_1\ f_2\ f_3 : \text{void}}\,(\texttt{cat})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : l \quad \Gamma' \mid \Gamma'' \vdash f_2 : l' \quad u \in o\delta(l) \wedge u\delta(l') \quad u\delta(l) \sqsubseteq u\delta(l') \quad \beta(l') \sqsubseteq \mathrm{WO}^-}{\Gamma \mid \Gamma'', f_2 : l' \sqcup l \vdash u.\texttt{mv}\ f_1\ f_2 : \text{void}}\,(\texttt{mv})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : l \quad \alpha(l) \sqsubseteq \mathrm{LC}^{n>0} \quad u \in u\delta(l)}{\Gamma \mid \Gamma', f_1 : \mathit{red}(l), f_2 : \mathit{dst}(\alpha(l), \beta(l), u : u) \sqcup \mathit{dst}(l) \vdash u.\texttt{copy}\ f_1\ f_2 : \text{void}}\,(\texttt{copy})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : l \quad \Gamma' \mid \Gamma'' \vdash f_2 : l' \quad u \in u\delta(l) \wedge u\delta(l') \quad \beta(l) \wedge \beta(l') \sqsubseteq \mathrm{WO}^+}{\Gamma \mid \Gamma'', f_3 : (\alpha(l) \sqcup \alpha(l'), \beta(l) \sqcup \beta(l), u : u) \sqcup l \sqcup l' \vdash u.\texttt{append}\ f_1\ f_2\ f_3 : \text{void}}\,(\texttt{append})$$

$$\frac{\Gamma \mid \Gamma' \vdash f_1 : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f_2 : l \vdash u.\texttt{move}\ f_1\ f_2 : \text{void}}\,(\texttt{move})$$

Figure 6.4: Typing rules

6.5 Downgrading and upgrading of policies

In this section we introduce several commands to manipulate file policies. Unlike previous commands which affect the files themselves, the commands introduced in this section affect the labels associated with the files. Such commands might downgrade or upgrade a file policy. Downgrading a policy, referred to as *declassification* in the literature, is the process of making the policy less restrictive, while upgrading a policy is the process of making the policy more restrictive. Although upgrading might occur implicitly as a result of information flow from a file with a less restrictive policy to a file with a more restrictive policy; performing it explicitly might be useful in some situations. For example, an owner might find out later that a file should not be written into it anymore, and allow it only to be read. On the other hand, downgrading cannot occur as a result of information flow. However, downgrading might be useful in some situations, for example, the owner of a file might find out later that a file need not be confidential anymore and allow it to be read.

Downgrading policies is a critical operation and leads to policy violation if it used improperly. For example, a user who is allowed to only write to a file, might downgrade the policy of a file to allow himself to read and write to the file. To prevent such violation of policies, we allow only owners of files to issue commands that manipulate policies. Therefore, only the owner of a file can change the policy associated with that file. Since a file might be associated with several policies that belong to different owners, we allow the commands that manipulate policies to affect only the policy that belongs to the owner who issues that command. Because only the effective policy will be taken into account when evaluating a label associated with the file, the other polices belonging to other owners will not be affected. That is, downgrading a single policy in a label consisting of multiple policies, will not change the overall policy of the label if there is another policy that is more restrictive than the downgraded policy.

As explained in the previous section, a label l might consist of a single policy (e.g. pl), or multiple policies (e.g. $pl_1, pl_2, pl_3, \ldots, pl_n$), where each policy belongs to an owner. Below we define functions useful for typing commands presented in the next section.

Definition 6.5.1. The function $sp(l, u, pl)$, takes a label l, a user u, and a policy pl, and swaps every policy in l owned by the user u with the policy pl.

$$sp(l, u, pl) = (\forall k \in l \wedge o\delta(k) = u \rightarrow swap(k, pl))$$

Definition 6.5.2. The function $ap(l, pl)$, takes a label l and a policy pl and appends the policy pl to the label l.

$$ap(l, pl) \rightarrow pl \sqcup l$$

Definition 6.5.3. The function $s\alpha(l, u, \alpha)$, takes a label l, a user u, and a security copy type α, and swaps every security copy type in every policy in l owned by the user u with the security copy type α.

$$s\alpha(l, u, \alpha) = (\forall k \in l \wedge o\delta(k) = u \rightarrow swap(\alpha(k), \alpha))$$

Definition 6.5.4. The function $s\beta(l, u, \beta)$, takes a label l, a user u, and a security access type β, and swaps every security access type in every policy in l owned by the user u with the security access type β.

$$s\beta(l, u, \beta) = (\forall k \in l \wedge o\delta(k) = u \rightarrow \mathit{swap}(\beta(k), \beta))$$

Definition 6.5.5. The function $s\delta(l, u, \delta)$, takes a label l, a user u, and an ownership and authorisation type δ, and swaps every ownership and authorisation type in every policy in l owned by the user u with the ownership and authorisation type δ.

$$s\delta(l, u, \delta) = (\forall k \in l \wedge o\delta(k) = u \rightarrow \mathit{swap}(\delta(k), \delta))$$

Definition 6.5.6. The function $ru(l, u_1, u_2)$, takes a label l and two users u_1 and u_2, and removes the user u_2 from the authorised users of every policy in the label l owned by the user u_1.

$$ru(l, u_1, u_2) = (\forall k \in l \wedge o\delta(k) = u_1 \wedge u_2 \in u\delta(k) \rightarrow \mathit{remove}(u_2, k))$$

Definition 6.5.7. The function $au(l, u_1, u_2)$, takes a label l and two users u_1 and u_2, and adds the user u_2 to the authorised users of every policy in the label l owned by the user u_1.

$$au(l, u_1, u_2) = (\forall k \in l \wedge o\delta(k) = u_1 \wedge u_2 \notin u\delta(k) \rightarrow \mathit{add}(u_2, k))$$

In the next section, we present the commands for manipulating file policies along with descriptions and their typing rules.

6.5.1 Language and typing rules

We extend the syntax of the language presented in the previous section with the following commands:

$$\begin{array}{rcl} \langle c \rangle & ::= & \texttt{chmod}\alpha\ \langle f \rangle\ \langle \alpha \rangle \mid \texttt{chmod}\beta\ \langle f \rangle\ \langle \beta \rangle \mid \texttt{chmod}\delta\ \langle f \rangle\ \langle \delta \rangle \mid \texttt{chmodp}\ \langle f \rangle\ \langle pl \rangle \mid \\ & \mid & \texttt{addp}\ \langle f \rangle\ \langle pl \rangle \mid \texttt{rmuser}\ \langle f \rangle\ \langle u \rangle \mid \texttt{adduser}\ \langle f \rangle\ \langle u \rangle \end{array}$$

The rest of the language remains unchanged. The additional commands presented above allow owners of files to manipulate their file policies in different ways. Owners can change a particular type of policy they own by issuing `chmod`α, `chmod`β, or `chmod`δ. Owners can also change the whole policy by issuing the command `chmodp`, or can add a new policy by issuing the command `addp`. Finally, owners can add or remove users from the policies they own by issuing the commands `adduser` and `rmuser`, respectively. Below we give a brief description of each command with its typing rule.

$\texttt{chmod}\alpha$: The command $\texttt{chmod}\alpha\ f\ \alpha$ changes the security copy type α of the policies owned by the user who issues the command that exist in the label associated with the file f. To successfully execute the command `chmod` $f\ \alpha$, the following constraints must be satisfied: *a*) The file f must already exist in the system. *b*) The user who issues the command must be an owner of a policy in the label associated with the file f. *c*) After executing the command, the security copy of the policies owned by the user who issued the command must be changed to the new type α that is provided to the command $\texttt{chmod}\alpha\ f\ \alpha$. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f : s\alpha(l, u, \alpha) \vdash u.\texttt{chmod}\alpha\ f\ \alpha : \text{void}}\ (\texttt{chmod}\alpha)$$

$\texttt{chmod}\beta$: The command $\texttt{chmod}\beta\ f\ \beta$ changes the security access type β of the policies owned by the user who issues the command that exist in the label associated with the file f. To successfully execute the command, the following constraints must be satisfied: *a*) The file f must already exist in the system. *b*) The user who issues the command must be an owner of a policy in the label associated with the file f. *c*) After executing the command, the security access type of the policies owned by the user who issued the command must be changed to the new type β that is provided to the command $\texttt{chmod}\beta\ f\ \beta$. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f : s\beta(l, u, \beta) \vdash u.\texttt{chmod}\beta\ f\ \beta : \text{void}}\ (\texttt{chmod}\beta)$$

$\texttt{chmod}\delta$: The command $\texttt{chmod}\delta\ f\ \delta$ changes the ownership and authorisation type δ of the policies owned by the user who issues the command that exist in the label associated with the file f. To successfully execute the command, the following constraints must be satisfied: *a*) The file f must already exist in the system. *b*) The user who issues the command must be an owner of a policy in the label associated with the file f. *c*) After executing the command, the ownership and authorisation type of the policies owned by the user who issued the command must be changed to the new type δ that is provided to the command $\texttt{chmod}\delta\ f\ \delta$. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f : s\delta(l, u, \delta) \vdash u.\texttt{chmod}\delta\ f\ \delta : \text{void}}\ (\texttt{chmod}\delta)$$

`chmodp`: The command `chmodp` $f\ pl$ changes the all policies owned by the user who issues the command that exist in the label associated with the file f. To successfully execute the command, the following constraints must be satisfied: *a*) The file f must already exist in the system. *b*) The user who issues the command must be an owner of a policy in the label associated with the file f. *c*) After executing the command, the policies owned by the user who issued the command must be changed to the new policy pl that is provided to the command `chmodp` f pl. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f : sp(l, u, pl) \vdash u.\texttt{chmodp}\ f\ pl : \text{void}}\ (\texttt{chmodp})$$

`addp`: The command `addp` f pl adds a new policy by the user who issues the command to the label associated with the file f. To successfully execute the command, the following constraints must be satisfied: *a*) The file f must already exist in the system. *b*) The user who issues the command must be an owner of a policy in the label associated with the file f. *c*) After executing the command, the policy that is provided to the command `addp` f pl must be appended to the label associated with the file f. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f : ap(l, pl) \vdash u.\texttt{addp}\ f\ pl : \text{void}}\ (\texttt{addp})$$

`adduser u`: The command `adduser` f u_1 adds a new authorised user to the policies owned by the user who issues the command that exist in the label associated with the file f. To successfully execute the command, the following constraints must be satisfied: *a*) The file f must already exist in the system. *b*) The user who issues the command must be an owner of a policy in the label associated with the file f. *c*) After executing the command, the user u_1 that is provided to the command `adduser` f u_1 must be appended to the authorised users of the policies owned by the user who issued the command. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f : au(l, u, u_1) \vdash u.\texttt{adduser}\ f\ u_1 : \text{void}}\ (\texttt{adduser})$$

`rmuser u`: The command `rmuser` f u_1 removes an existing authorised user from the policies owned by the user who issues the command that exist in the label associated with the file f. To successfully execute the command, the following constraints must be satisfied: *a*) The file f must already exist in the system. *b*) The user who issues the command must be an owner of a policy in the label associated with the file f. *c*) After executing the command, the user u_1 that is provided to the command `rmuser` f u_1 must be removed from the authorised users of the policies owned by the user who issued the command. This leads to the following typing rule:

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f : ru(l, u, u_1) \vdash u.\texttt{rmuser}\ f\ u_1 : \text{void}}\ (\texttt{rmuser})$$

The typing rules presented above are depicted in Figure 6.5; which is an extension to the typing rules depicted in Figure 6.4. In the next section we revise the typing algorithm to reflect the new version of the type system shown in Figures 6.4 and 6.5.

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f : s\alpha(l, u, \alpha) \vdash u.\texttt{chmod}\alpha\ f\ \alpha : \text{void}}\ (\texttt{chmod}\alpha)$$

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f : s\beta(l, u, \beta) \vdash u.\texttt{chmod}\beta\ f\ \beta : \text{void}}\ (\texttt{chmod}\beta)$$

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f : s\delta(l, u, \delta) \vdash u.\texttt{chmod}\delta\ f\ \delta : \text{void}}\ (\texttt{chmod}\delta)$$

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f : sp(l, u, pl) \vdash u.\texttt{chmodp}\ f\ pl : \text{void}}\ (\texttt{chmodp})$$

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f : ap(l, pl) \vdash u.\texttt{addp}\ f\ pl : \text{void}}\ (\texttt{addp})$$

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f : au(l, u, u_1) \vdash u.\texttt{adduser}\ f\ u_1 : \text{void}}\ (\texttt{adduser})$$

$$\frac{\Gamma \mid \Gamma' \vdash f : l \quad u \in o\delta(l)}{\Gamma \mid \Gamma', f : ru(l, u, u_1) \vdash u.\texttt{rmuser}\ f\ u_1 : \text{void}}\ (\texttt{rmuser})$$

Figure 6.5: Typing rules for changing policies

6.5.2 Typing algorithm

In this section we present a type inference algorithm $\mathcal{T}$ for typing phrases according to the new type system depicted in Figures 6.4 and 6.5. In addition to the functions $check, less, red$ and dst defined in the previous typing algorithm, we define the following functions. If $\tau = l$, then $\pi_1(\tau) = \alpha(l), \pi_2(\tau) = \beta(l), \pi_3(\tau) = o\delta(l)$, and $\pi_4(\tau) = u\delta(l)$. $auth(u, \tau)$ returns true if $u \in \pi_4(\tau)$, and $own(u, \tau)$ returns true if $u \in \pi_3(\tau)$. If $\tau = l$ and $\tau' = l'$, then $lub(\tau, \tau') = l \sqcup l'$. Using these functions, we can now define the type inference algorithm $\mathcal{T}$ as follows:

The Type Reconstruction Algorithm $\mathcal{T}$:

$$\mathcal{T}(A, e) = (\tau, A')$$

where:

1. If e is the filename f, and $f : \alpha \in A$ then $\tau = \alpha$, $A' = A \setminus \{f : \alpha\}$.

2. If e is a sequence of commands, $u.c; u.cs$ let

$$\begin{aligned} (\alpha, A_1) &= \mathcal{T}(A, c) \\ check(\alpha, \text{void}) & \\ (\beta, A_2) &= \mathcal{T}(A_1, cs) \\ check(\beta, \text{void}) & \end{aligned}$$

   then $\tau = \text{void}, A' = A_2$.

3. If e is the cp command, $u.\texttt{cp}\ f_1\ f_2$ let

$$\begin{aligned} (\alpha, A_1) &= \mathcal{T}(A, f_1) \\ less(\pi_1(\alpha), \text{LC}^{n>0}) & \\ (\beta, A_2) &= \mathcal{T}(A_1, f_2) \\ auth(u, \alpha) & \\ auth(u, \beta) & \\ less(\pi_2(\beta), \text{WO}^-) & \\ less(\pi_4(\alpha), \pi_4(\beta)) & \end{aligned}$$

   then if $f_1, f_2 \notin A_2$, then $\tau = \text{void}, A' = A_2 \cup \{f_1 : \mathit{red}(\alpha), f_2 : \mathit{lub}(\mathit{dst}(\alpha), \beta)\}$.

4. If e is the rm command, $u.\texttt{rm}\ f$ let

$$\begin{aligned} (\alpha, A_1) &= \mathcal{T}(A, f) \\ own(u, \alpha) & \end{aligned}$$

   then $\tau = \text{void}, A' = A_1$.

5. If e is the mkf command, $u.\texttt{mkf}\ f\ \alpha\ \beta$, then if $f \notin A$, then $\tau = \text{void}, A' = A \cup \{f : \alpha, \beta, u : u\}$.

6. If e is the rd command, $u.\texttt{rd}\ f$ let

$$\begin{aligned} (\alpha, A_1) &= \mathcal{T}(A, f) \\ less(\pi_2(\alpha), \text{RO}) & \\ auth(u, \alpha) & \end{aligned}$$

   then $\tau = \text{void}, A' = A_1$.

7. If e is the cat command, u.`cat` f_1 f_2 f_3 let

$$\begin{array}{lcl}
(\alpha, A_1) & = & \mathcal{T}(A, f_1) \\
(\beta, A_2) & = & \mathcal{T}(A_1, f_2) \\
(\delta, A_3) & = & \mathcal{T}(A_2, f_3) \\
auth(u, \alpha) & & \\
auth(u, \beta) & & \\
auth(u, \delta) & & \\
less(\pi_4(\alpha), \pi_4(\delta)) & & \\
less(\pi_4(\beta), \pi_4(\delta)) & & \\
less(\pi_2(\alpha), \mathrm{WO}^+) & & \\
less(\pi_2(\beta), \mathrm{WO}^+) & & \\
less(\pi_2(\delta), \mathrm{WO}^-) & &
\end{array}$$

then if $f_3 \notin A_3$, then $\tau = \text{void}, A' = A_3 \cup \{f_3 : lub(\alpha, \beta, \delta)\}$.

8. If e is the mv command, u.`mv` f_1 f_2 let

$$\begin{array}{lcl}
(\alpha, A_1) & = & \mathcal{T}(A, f_1) \\
(\beta, A_2) & = & \mathcal{T}(A_1, f_2) \\
less(\pi_2(\beta), \mathrm{WO}^-) & & \\
own(u, \alpha) & & \\
auth(u, \beta) & & \\
less(\pi_4(\alpha), \pi_4(\beta)) & &
\end{array}$$

then if $f_2 \notin A_2$, then $\tau = \text{void}, A' = A_2 \cup \{f_2 : lub(\alpha, \beta)\}$.

9. If e is the copy command, u.`copy` f_1 f_2 let

$$\begin{array}{lcl}
(\alpha, A_1) & = & \mathcal{T}(A, f_1) \\
less(\pi_1(\alpha), \mathrm{LC}^{n>0}) & & \\
auth(u, \alpha) & &
\end{array}$$

then if $f_2 \notin A$, then $\tau = \text{void}, A' = A_1 \cup \{f_1 : red(\alpha), f_2 : lub(dst(\pi_1(\alpha), \pi_2(\beta), u : u), dst(\alpha))\}$.

10. If e is the append command, $u.$`append` $f_1\ f_2\ f_3$ let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, f_1) \\ (\beta, A_2) & = & \mathcal{T}(A_1, f_2) \\ auth(u, \alpha) & & \\ auth(u, \beta) & & \\ less(\pi_2(\alpha), \mathrm{WO}^+) & & \\ less(\pi_2(\beta), \mathrm{WO}^+) & & \end{array}$$

then if $f_3 \notin A$, then $\tau = \mathrm{void}, A' = A_2 \cup \{f_3 : lub((lub(\pi_1(\alpha), \pi_1(\beta)), lub(\pi_2(\alpha), \pi_2(\beta)), u : u), \alpha, \beta)\}$.

11. If e is the move command, `move` $f_1\ f_2$ let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, f_1) \\ own(u, \alpha) & & \end{array}$$

then if $f_2 \notin A$, then $\tau = \mathrm{void}, A' = A_1 \cup \{f_2 : \alpha\}$.

12. if e is the chmodα command, $u.$`chmod`$\alpha\ f\ \alpha$ let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, f) \\ own(u, \alpha) & & \end{array}$$

then if $f \notin A_1$, then $\tau = \mathrm{void}, A' = A_1 \cup \{f : s\alpha(\alpha, u, \alpha)\}$

13. if e is the chmodβ command, $u.$`chmod`$\beta\ f\ \beta$ let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, f) \\ own(u, \alpha) & & \end{array}$$

then if $f \notin A_1$, then $\tau = \mathrm{void}, A' = A_1 \cup \{f : s\beta(\alpha, u, \beta)\}$

14. if e is the chmodδ command, $u.$`chmod`$\delta\ f\ \delta$ let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, f) \\ own(u, \alpha) & & \end{array}$$

then if $f \notin A_1$, then $\tau = \mathrm{void}, A' = A_1 \cup \{f : s\delta(\alpha, u, \delta)\}$

15. if e is the chmod$p$ command, $u.$`chmodp` $f\ pl$ let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, f) \\ own(u, \alpha) & & \end{array}$$

then if $f \notin A_1$, then $\tau = \mathrm{void}, A' = A_1 \cup \{f : sp(\alpha, u, pl)\}$

16. if e is the addp command, u.`addp` f pl let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, f) \\ own(u, \alpha) & & \end{array}$$

then if $f \notin A_1$, then $\tau = \text{void}, A' = A_1 \cup \{f : ap(\alpha, pl)\}$

17. if e is the adduser command, u.`adduser` f u_1 let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, f) \\ own(u, \alpha) & & \end{array}$$

then if $f \notin A_1$, then $\tau = \text{void}, A' = A_1 \cup \{f : au(\alpha, u, u_1)\}$

18. if e is the rmuser command, u.`rmuser` f u_1 let

$$\begin{array}{lcl} (\alpha, A_1) & = & \mathcal{T}(A, f) \\ own(u, \alpha) & & \end{array}$$

then if $f \notin A_1$, then $\tau = \text{void}, A' = A_1 \cup \{f : ru(\alpha, u, u_1)\}$

6.6 Discussion

In the last revision of the type system where types of files are labels and the typing rules are extended to control which operations can be performed on which label of the file and by whom; various policies identified in Chapter 4 can be specified and enforced. Particularly, the different types of access and propagation in a shared memory style can be enforced. For example, a file with a label such as $\{\text{NC}, \text{RO}, Alice : Bob\}$ describes a file that is shared as OneToOne in a shared memory and the file can be read only once. While a file with a label such as $\{\text{UC}, \text{WO+}, Alice : Bob, Carol, Dave\}$ describes a file that is shared as OneToGroup in a shared memory and the file can be appended to it only. Other types of propagation such as Group, GroupToOne, and ManyToOne are built of atomic types. For example, Group describes a situation where each one of the group is sharing his file as OneToGroup. That is, if $Alice, Bob$, and $Carol$ want to share their files as Group, then any file created by each of them should specify the others as authorised users. This will make more sense if we extend our system with directories which we aim for in future work. For example, a directory of type GroupToOne such as $\{Alice, Bob, Carol\}To\{Dave\}$, should only store files of types $\{Alice : Dave\}$, $\{Bob : Dave\}$, or $\{Carol : Dave\}$. By having directories in the system, a distinction between publishing and sharing in static, dynamic, and transfer modes becomes obvious. That is copying a file from one directory to another by `cp` or `copy` commands is publishing or sharing in a static mode, while moving a file from one directory to another by `mv` or `move` commands is publishing or sharing in transfer mode. Publishing or sharing in dynamic mode, which is another extension we aim for in future work, can be achieved in the same way as in the static mode, except that a reference to a file is copied from one directory to another rather than the file itself.

The developed type system in the previous chapter and the extensions discussed in this chapter are focused on enforcing the different types of access and propagation in a shared memory style, represented as a file system. Other possible future extensions are to enforce restrictions over types of access other than limiting the number of times a file can be read, such as limiting the period of time, the location and the specific time for a particular access type to be exercised, and enforcing these policies in a distributed memory style where files are moved from the owner device to the recipient device to be accessed locally rather than stored in a particular place that must be accessed by all recipients.

The type system developed in this thesis is not confined to enforcing the policies identified in Chapter 4. Other policies that we have not looked at in this thesis can be enforced similarly with the basic idea of the type system which is based on resource consumption and intercepting commands. For example, the Bell-LaPadula model of multi-level security can be enforced by our type system if we associate users and files with security levels such as top secret, secret, confidential, and unclassified. The simple security property (no read up) which requires that a user at a given security level may not read a file at a higher security level, can be achieved by extending the typing rule for $u.\mathtt{rd}\ f$ command to check for $l(f) \sqsubseteq l(u)$, that is the security level of the user is at least as restrictive as the security level of the file. The star property (no write down) which requires that a user at a given security level may not write to file at a lower security level, can be achieved by extending the typing rule for $u.\mathtt{mkf} f\ l$ command to check for $l(u) \sqsubseteq l$, that is the security level assigned to the file to be created is at least as restrictive as the security level of the user creating the file. The typing rules for other commands that writes to files such as `cat`, `mv` and `append`, need not to be extended. This is because they change the security level of files only to be more restrictive, and thus the star property will be satisfied.

The policies enforced by our type system are a sort of discretionary access control, in the sense that owners can specify any policies they prefer to their files. For example, the command $u.\mathtt{mkf}\ f\ l$ allows the user u to create a file f associated with label l. The typing rule for the command $u.\mathtt{mkf}$ has no constraint over the label to be associated with the file. However, mandatory access control where system-wide policies must be enforced and owners have no discretion over file policies might be useful in some situations. For example, in an organisation, *Alice* might only be allowed to write reports and share them only with her manager *Bob* who can only read them. Because these reports are confidential, the organisation need to ensure that they cannot be shared accidentally with anyone else except *Bob*. While *Alice* can create these reports with the label $\{\text{UC}, \text{RO}, Alice : Bob\}$ to ensure that only *Bob* can read them, there is a chance that *Alice* might accidentally create them with another label that allows others to read them. To enforce this sort of mandatory access control in our type system, we need to associate users in the system with labels that are identical to the labels associated with the files. User labels represent the maximum policy which can be associated with the files they create. Therefore, mandatory access control can be simply enforced by extending only the typing rule for the command $u.\texttt{mkf}\ f\ l$ to check for $l \sqsubseteq u(l)$ which ensures that the label to be associated with the created file is less or equal to the maximum policy the user can specify.

Our type system enforces both access control and information flow policies. Access control is enforced by checking which operations can be performed on which file and by whom, while information flow is enforced by tracking file policies and allowing them to be changed only to more restrictive policies. The flow policies enforced by our type system lie somewhere between the flow policies enforced by flow-insensitive type systems and flow-sensitive type systems. This is because of the following two reasons. Firstly, types of files in our system are not just security levels but they also represent permissions that dictate which operations are allowed to be performed on them. Information flow between files in our system can only occur by performing operations on these files and are allowed if and only if the operations are permitted by the types of files. In flow-insensitive type systems flow of information from f_1 to f_2 is allowed if and only if $T(f_1) \sqsubseteq T(f_2)$. That is, if $T(f_1) = \text{RO}$ and $T(f_2) = \text{NRW}$, then flow-insensitive type systems will allow information to flow from f_1 to f_2 because $\text{RO} \sqsubseteq \text{NRW}$. However, this will violate the policy of f_2 because the type of f_2 is NRW which requires that such a file cannot be read or written into it. More interestingly, if $T(f_1) = \text{NRW}$ and $T(f_2) = \text{RW}^-$, then flow-insensitive type systems prevent flow of information from f_1 to f_2 since $\text{NRW} \not\sqsubseteq \text{RW}^-$. However, such flow is allowed in our type system because the type of f_2 is RW^- which allows such a file to be overwritten. Secondly, our view is that each file is associated with a policy that must be enforced. A file must enforce its own policy and the policy of the information flowed into it. In flow-insensitive type systems the flow of information from a source to a destination causes the information to acquire the policy of the destination. On the other hand, in flow-sensitive type system the flow of information from a source to a destination causes the destination to acquire the policy of the source of the information. However, both ways will violate the file policies. For example, assume that $T(f_1) = \text{NRW}$ and $T(f_2) = \text{RW}^-$ and there is information flow from f_1 to f_2. If we let the information flowing to acquire the type of f_2, then f_1 can be read indirectly by reading f_2 where the type of f_1 is NRW which requires that such a file cannot be read. Now assume that $T(f_1) = \text{RO}$ and $T(f_2) = \text{WO}^-$ and there is information flow from f_1 to f_2. If we let f_2 acquires the type of the source of information, then f_2 can be read where the type of f_2 is WO^- which requires that such a file can be only written into but not read.

The flow policy enforced by our type system follows the idea of flow-insensitive type systems in that the flow of information must only result in a more restrictive type of information; as well as the idea of flow-sensitive type systems in that information can flow anywhere and the security types can be changed during computation. In such a way we may benefit from the restrictiveness of flow-insensitive type systems and the permissiveness of flow-sensitive type systems.

Our approach to extend file policies with ownership and authorisation information and represent them as a label is inspired by the work on DLM. However, our label structure, interpretation, and flow policy is different from DLM. In DLM, a policy in a label consists of two components which are an owner and a reader set, whereas a policy in our label consists of four components which are a security copy type, a security access type, an owner, and authorised users. Labels in DLM represent policies that dictate where the information can flow, whereas labels in our system represent permissions of files that dictate which operations can be performed and by whom. In DLM, the flow of information from a source to a destination causes the information to acquire the label of the destination, referred to as information relabeling. That is, it enforces the flow policy of flow-insensitive type systems. Such relabeling is only allowed if it is a restriction, that is the new label must only remove readers, add owners, or both. However, in our system such flow causes the destination label to change its label to enforce all the policies in both its label and the source label. This is similar to how derived values are treated in DLM. Therefore, relabeling occurs to the destination rather than to the information flowing to that destination. Unlike DLM, relabeling in our system is only allowed if the operation that causes such relabeling is permitted by both labels of the source and destination, and the authorised users of the destination label is a subset of the authorised user of the source label. A major difference between our work and the work on DLM is the idea of consumption of resource and intercepting commands that manipulate files. In DLM, the flow of information from a source to a destination causes a copy of the information stored in the source to be assigned to the destination. However, in our work such a flow causes the source to be consumed and stored in the destination unless otherwise the source is explicitly copied. DLM is a general model that only restricts how information can flow between different parts of the system, whereas our work is focused on file sharing that intercepts each command to be performed on files and checks for access control requirements and enforces the information flow requirements. In DLM, users of a label are considered either readers or writers of the information of that label. Whereas in our work, users of a label are considered authorised to perform operations that are specified by the label, which might not allow them to read or write to the information of that label.

The current labels associated with files divide users into owners and authorised users. Each label assigns the same permissions to all authorised users. This might not be desirable in situations where different users require different permissions for the same file. For example, if *Alice* needs to read the file f_1 and *Bob* needs to write to the file f_1, then our labels cannot specify this policy. That is, our label cannot grant different permissions to different users for the same file. In fact, in Unix-like file systems, where traditional file permissions model is used, each file can grant different permissions to three different types of users: owner, group, and others. Moreover, Access Control List (ACL) can be used as an extension to traditional file permissions model to avoid its limitations, and allow permissions to be granted to individual users or groups even if these do not correspond to the original owner or the owning group.

However, our label is useful in situations where all authorised users need to have the same permissions for the same file. For situations where two groups of users need to have different permissions for the same file, then two linked copies of the file must exist where each copy is associated with a label that specifies the permissions needed for one group. For example, if there are two groups of users $group_1$ and $group_2$, where $group_1$ need to read the file f_1 and $group_2$ need to write to the file f_1. Then, a reference to the file f_1 should be copied into f_2, and f_1 should be associated with a label to specify the permissions for $group_1$, and f_2 should be associated with a label to specify the permissions for $group_2$. In this way, any changes made to f_2 will be reflected into f_1, and thus, it will have the same effect as assigning different permissions to different users for the same file.

Implementing the type system developed in this thesis in a real file system is one thing that we are aiming for in future work. Such implementation is useful to show the practicality of our approach to prevent accidental misuse of shared files, and also to obtain feedback on using the system for possible improvements. Various choices for implementation have to be made; for example, different choices available to associate types with files. Firstly, they can be attached to files as described in the file system δ. Secondly, they can be stored along with file names in a different location internal to the file system. Thirdly, they can be stored with file names in a different location external to the file system. Although each of them might have advantages and disadvantages, the role of the type system will be the same which is to intercept each command to be performed on files and fetch the types of these files to check for access control and information flow requirements.

6.7 Summary

In this chapter we have looked at possible future extensions to the type system presented in Chapter 5. In particular, we showed that the type system can be easily extended to regulate other operations than *copy* in order to enforce the various policies identified in Chapter 4. We have taken a significant step towards realising these extensions. We began by defining additional security types to control *read* and *write* operations, which we refer to as security access types. We showed that if files were associated only with the security access types, then the same typing rules presented in Chapter 5 with additional constraints can be used to control the access to *read* and *write* operations and the flow caused by all operations. Then, we defined security types of files as pairs that consist of a security copy type and a security access type. The former type represents a policy to control the access and flow of *copy* operations; and the latter type represents a policy to control the access and flow of *read* and *write* operations. We extended the type system to enforce these policies along with a typing algorithm. The extended type system controls the access to *copy*, *read* and *write* operations and the flow caused by all operations. Next, we defined security types of files as labels that not only consist of a security copy type and a security access type, but also of ownership authorisation information. The ownership and authorisation information in a label indicates the owners and the authorised users of a file associated with the label. Such labels represent policies to specify which operations can be performed on which types of files and by whom. Based on the definition of labels, we extended the type system to not only control the access and flow of operations but also control which user can perform these operations. Finally, we extended the commands in our language to include commands that manipulate file policies. We extended the type system with typing rules for these commands along with a typing algorithm for typing phrases in accordance with the last extension of the type system.

Chapter 7

Conclusion

File sharing has been a topic of interest in computer science, ever since files were created. One of the most challenging problems researchers face is protecting the shared files from various attacks that violate their confidentiality, integrity and availability. These attacks can be launched by unauthorised users, referred to as external threats, or by authorised users, referred to as insider threats. In this thesis, we have investigated the insider threat problem with respect to file sharing and developed a novel approach to preventing accidental threats to the shared files.

There exists a large body of work in the literature on addressing the insider threat problem. The problem of the insider threat is not only confined to attacks on shared files, but rather it encompasses various types of attack that target different assets of an organisation. As a result of the broad scope of the problem, various definitions of the insider threat problem exist. Researchers have described the insider threat problem by defining who the insiders are and what threats they constitute. Their definitions contradict one another; and what is considered an insider for some researchers might be considered an outsider for others. Also, they have generally always described insider threats based on a definition of the insider, such that the insider threats are the damages caused to an organisation by an insider. Due to the fact that there is no clear definition of the problem, little progress has been made in addressing the insider threat problem. From our point of view, better progress can be made if the problem is classified into several categories which can be defined, studied and solved independently and which later can be combined to solve the problem as a whole. Therefore, we proposed an approach to classifying the insider problem into different categories and providing precise definitions of who the insider is and what is the insider problem. Based on the proposed classification, we defined our class of insider problem; namely, preventing confidentiality and integrity attacks on sensitive files by recipients during the activity of file sharing. Although the insiders and the assets that we need to protect are clearly identified in our class of insider problem, the attacks and the activities are still vague. Files can be shared and attacked by insiders in different ways which must be identified. Different types of attack by insiders require different types of protection. Claiming that a particular protection mechanism can protect file confidentiality is not enough. Instead, one should claim that a particular protection mechanism can protect file confidentiality under specific types of attack. Identifying these types of attack makes it clear which protection mechanism we need to develop, and allow us to validate it against the types of attack it claims to prevent. Therefore, we investigated the different types of misuse of the shared files that can be performed by insiders during the activity of file sharing, and we characterised the protection required against them. We focused in this thesis on the protection required to prevent accidental misuse that affects the confidentiality and integrity of files by trusted insiders. This is because files can only be entirely protected if shared with trusted insiders. System vulnerabilities and the analogue hole problem have made protection against untrusted insider unfeasible. Untrusted malicious insiders will always find a way to bypass the protection mechanism in place. Also, accidental misuse of the shared files by insiders is a highly cited problem in the literature [34, 4, 54], and if this cannot be solved, then neither can deliberate misuse.

Protecting the shared files is a topic that has been studied in two different fields with different interests, namely, information sharing and security. The former focuses on facilitating information sharing and provides sharing tools that are suitable for various sharing tasks but not secure. The latter focuses on securing information sharing and provides sharing tools that are secure but not suitable for every sharing task. Considering both fields will help us to design a protection mechanism that will not only protect the shared files against accidental misuse by insiders; but will also not interfere with people's practices of file sharing. Therefore, in addition to identifying the misuse we need to prevent, we investigate how the activity of file sharing can be performed. We characterised the activity of file sharing according to how files can be propagated from owners to recipients and how files can be accessed by the recipients after their propagation. Based on this characterisation, we defined a framework that classifies the activity of file sharing into different categories. Each category specifies how files should be propagated and accessed after their propagation. We showed that these categories can be thought of as policies that, if enforced, allow the provision of various types of protection against accidental misuse.

We enforced these policies by the use of language based techniques; and designed a language to manipulate files and specify their policies in a file system, and a type system that enforces these policies. In the file system, policies are represented as security types which are associated with files, and programs are set of operations to be performed on files. The security types represent both access control and information flow policies. They represent access control polices as they dictate which operations are allowed to be performed; and represent information flow policies as they dictate where the information can flow. The role of the type system is to intercept each command to be performed on files, and enforce the access control and information flow policies of these files. That is, the type system will first check whether or not the operations to be performed on files are allowed by the types of the files, and secondly will check whether or not the information flow between files caused by the command satisfies the flow policies of the files.

As a starting point, we focused on enforcing a particular constraint of the policies; namely, limiting the number of times a file can be read. We achieved this by limiting the number of copies of a file that can be produced, and by the notion of resource consumption; that is a file is a resource which must be consumed when it is used unless if it is explicitly copied. Therefore, we define security types to control the access to copy operations and the flow caused by all operations including copy, such that the copy policies of files are not violated. However, other constraints can be enforced similarly by controlling the access to and the flow caused by other operations, as we discussed in Chapter 6. We proved the soundness of the developed type system that enforces these policies and defined a type reconstruction algorithm and proved its soundness and completeness.

The approach taken in this thesis to tackle accidental insider threats to file sharing is not yet completed. However, we have developed the basic elements of the system which are capable of showing the usefulness and practicality of our approach to tackle accidental insider threats to file sharing, and which can be built upon easily to realise the complete picture of our approach. Various extensions are left for future work as pointed out in Chapter 6. Some of them are discussed in detail with minimal work left to complete them, while others require further investigation. Implementing the system developed in this thesis is one of the things that we aim for in future work. Such implementation is useful to obtain feedback from users using the system about our approach that could be used to improve and refine the current system.

In conclusion, this thesis has proposed a novel approach to prevent accidental insider threats to file sharing. I hope that this thesis will inspire current researchers who study insider threats, file sharing, and language-based security, to work together towards developing a common secure language that allows secure file sharing against accidental insider threats, by refining and extending the approach taken in this thesis to suit their purpose and need.

Bibliography

[1] Abadi, M. (1999). Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786. 48, 53

[2] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. (2002). Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *PROCEEDINGS OF THE 5TH SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION (OSDI*, pages 1–14. 92

[3] Ahern, S., Eckles, D., Good, N. S., King, S., Naaman, M., and Nair, R. (2007). Overexposed?: privacy patterns and considerations in online and mobile photo sharing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 357–366, New York, NY, USA. ACM. 15, 16

[4] AlgoSec (2013). The state of network security 2013: Attitudes and opinions. `http://www.algosec.com/resources/files/Specials/Survey%20files/State%20of%20Network%20Security%202013_Final%20Report.pdf`. 8, 187

[5] Alhazmi, O. H., Malaiya, Y. K., and Ray, I. (2004). Vulnerabilities in major operating systems. Technical report, Department of Computer Science, Colorado State University. 6

[6] Anderson, R. and Brackney, R. (2004). Understanding the insider threat. In *Proceedings of a March 2004 Workshop. Prepared for the Advanced Research and Development Activity (ARDA). http://www. rand. org/publications/CF/CF196.* 34, 35

[7] Anderson, T. E., Dahlin, M. D., Neefe, J. M., Patterson, D. A., Roselli, D. S., and Wang, R. Y. (1995). Serverless network file systems. *SIGOPS Oper. Syst. Rev.*, 29(5):109–126. 92

[8] Arnab, A. and Hutchison, A. (2007). Persistent access control: a formal model for

drm. In *Proceedings of the 2007 ACM workshop on Digital Rights Management*, DRM '07, pages 41–53, New York, NY, USA. ACM. 42

[9] Arsenova, E. (n.d). Technical aspects of digital rights management. 45

[10] Baliello, C., Basso, A., Giusto, C. D., Khalil, H., and Machancoses, D. (2002). Kerberos protocol: an overview Distributed Systems. 31

[11] Bellovin, S. M. (2008). The Insider Attack Problem Nature and Scope. In Stolfo, S. J., Bellovin, S. M., Keromytis, A., Hershkop, S., Smith, S. W., and Sinclair, S., editors, *Insider Attack and Cyber Security - Beyond the Hacker*, volume 39 of *Advances in Information Security*. Springer. 36

[12] Bhatt, S., Sion, R., and Carbunar, B. (2009). A personal mobile drm manager for smartphones. *Computers & Security*, 28(6):327–340. 6, 7

[13] Birgisson, A., Russo, A., and Sabelfeld, A. (2010). Unifying facets of information integrity. In Jha, S. and Mathuria, A., editors, *ICISS*, volume 6503 of *Lecture Notes in Computer Science*, pages 48–65. Springer. 49

[14] Bishop, M. (2005). Position: "insider" is relative. In *Proceedings of the 2005 workshop on New security paradigms*, NSPW '05, pages 77–78, New York, NY, USA. ACM. 34

[15] Bishop, M., Engle, S., Peisert, S., Whalen, S., and Gates, C. (2009). Case studies of an insider framework. page 817. 35

[16] Bishop, M. and Gates, C. (2008). Defining the insider threat. pages 1–3, New York, NY, USA. ACM. 35

[17] Boudol, G. (2008). Secure information flow as a safety property. In Degano, P., Guttman, J. D., and Martinelli, F., editors, *Formal Aspects in Security and Trust*, volume 5491 of *Lecture Notes in Computer Science*, pages 20–34. Springer. 50

[18] Brown, B., Sellen, A. J., and Geelhoed, E. (2001). In *Proceedings of the seventh conference on European Conference on Computer Supported Cooperative Work*, ECSCW'01, pages 179–198, Norwell, MA, USA. Kluwer Academic Publishers. 15, 16

[19] CCITT (Consultative Committee on International Telegraphy and Telephony) (1991). *Recommendation X.800: Security Architecture for Open Systems Interconnection for CCITT Applications*. 29

[20] CERT (2013). The CERT Insider Threat Center @ONLINE. 35

[21] Chaudhuri, A. and Abadi, M. (2006). Secrecy by typing and file-access control. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 112–123. IEEE. 53

[22] Chinchani, R., Iyer, A., Ngo, H. Q., and Upadhyaya, S. (2005). Towards a theory of insider threat assessment. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, pages 108–117, Washington, DC, USA. IEEE Computer Society. 34

[23] Chothia, T., Duggan, D., and Vitek, J. (2003). Type-based distributed access control. In *In Proc. IEEE Computer Security Foundations Workshop*, pages 170–186. IEEE. 51

[24] Christensen, S. (Accessed on [20/8/2013]). Introduction to file sharing services: An it-forensic examination of p2p clients. 17, 18

[25] Clark, D. D. and Wilson, D. R. (1987). A Comparison of Commercial and Military Computer Security Policies. In *1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press. 41

[26] Dalal, B., Nelson, L., Smetters, D., Good, N., and Elliot, A. (2008). Ad-hoc guesting: when exceptions are the rule. In *Proceedings of the 1st Conference on Usability, Psychology, and Security*, UPSEC'08, pages 9:1–9:5, Berkeley, CA, USA. USENIX Association. 15, 22, 24, 27

[27] Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA. ACM. 103

[28] Denning, D. E. (1976). A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243. 48

[29] Denning, D. E. and Denning, P. J. (1977). Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513. 48

[30] Dezani-Ciancaglini, M. and De'Liguoro, U. (2010). Sessions and session types: An overview. In *Proceedings of the 6th International Conference on Web Services and Formal Methods*, WS-FM'09, pages 1–28, Berlin, Heidelberg. Springer-Verlag. 55

[31] DRM, A. `http://www.adobe.com/manufacturing/resources/drm/`. [Accessed: 2013-11-28]. 5

[32] DRM, A. F. http://www.apple.com/itunes. [Accessed: 2013-11-28]. 5

[33] Ernst, M. D. (2003). Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR. 48

[34] Europe, I. and PwC (2013). 2013 information security breaches survey. Technical report, Department for Business, Innovation & Skills. 8, 187

[35] Ferraiolo, D. and Kuhn, R. (1992). Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference*. 43

[36] Freier, A. O., Kariton, P., and Kocher, P. C. (1996). The SSL protocol: Version 3.0. Internet draft, Netscape Communications. 31

[37] Goguen, J. A. and Meseguer, J. (1982). Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. 49

[38] Good, N. S. and Krekelberg, A. (2003). Usability and privacy: a study of kazaa p2p file-sharing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '03, pages 137–144, New York, NY, USA. ACM. 15, 16

[39] Goudar, R. and More, P. (2011). Multilayer Security Mechanism in Computer Network. *International Journal of Computer Networks and Wireless Communications (IJCNWC)*, 1(1). 31

[40] Haber, S., Horne, B., Pato, J., Sander, T., and Tarjan, R. E. (2003). If piracy is the problem, is drm the answer? In Becker, E., Buhse, W., Gnnewig, D., and Rump, N., editors, *Digital Rights Management*, volume 2770 of *Lecture Notes in Computer Science*, pages 224–233. Springer. 6

[41] Hamzeh, K., Pall, G. S., Verthein, W., Taarud, J., Little, W. A., and Zorn, G. (1999). Point-to-point tunneling protocol (PPTP). Internet RFC 2637. 31

[42] Harinarayana, N. S., Somu, C. S., and Sunil, M. V. (2009). Digital rights management in digital libraries: An introduction totechnology, effects and the available open source tools. In *7th International CALIBER-2009, Pondicherry University, Puducherry, February 25-27, 2009*. 45

[43] Harris, S. (2002). *Mike Meyers' Cissp(r) Certification Passport*. McGraw-Hill Prof Med/Tech, 2002. 39, 40, 41, 42, 43

[44] Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. (1976). Protection in operating systems. *Commun. ACM*, 19(8):461–471. 47

[45] Hart, M., Johnson, R., and Stent, A. (2006). More content-less control: Access control in the web 2.0. *Control*, pages 1–3. 23

[46] Hauser, T. and Wenz, C. (2003). Drm under attack: Weaknesses in existing systems. In Becker, E., Buhse, W., Gnnewig, D., and Rump, N., editors, *Digital Rights Management*, volume 2770 of *Lecture Notes in Computer Science*, pages 206–223. Springer. 5, 6

[47] Hedin, D. and Sabelfeld, A. (2011). A perspective on information-flow control. 49

[48] Honda, K., Vasconcelos, V. T., and Kubo, M. (1998). Language primitives and type discipline for structured communication-based programming. In *In ESOP?98, volume 1381 of LNCS*, pages 122–138. Springer-Verlag. 54, 55

[49] Howard, J. H. (1988). An overview of the andrew file system. In *in Winter 1988 USENIX Conference Proceedings*, pages 23–26. 92

[50] Hunker, J. (2008). Taking Stock and Looking Forward - An Outsider's Perspective on the Insider Threat. In Stolfo, S. J., Bellovin, S. M., Keromytis, A., Hershkop, S., Smith, S. W., and Sinclair, S., editors, *Insider Attack and Cyber Security - Beyond the Hacker*, volume 39 of *Advances in Information Security*. Springer. 34

[51] Hunker, J. and Probst, C. W. (2011). Insiders and insider threats: An overview of definitions and mitigation techniques. *Jounral of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 2(1):4–27. 35, 36, 38, 59

[52] Hunt, S. and Sands, D. (2006). On flow-sensitive security types. *SIGPLAN Not.*, 41(1):79–90. 48, 50

[53] IBM (Accessed on [20/8/2013]). The floppy disk. 17

[54] Institute, P. (2012). The human factor in data protection. `http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt_trend-micro_ponemon-survey-2012.pdf`. 8, 187

[55] Institute, S. E. (2011). 2011 CyberSecurity Watch Survey. Software Engineering Institute, Carnegie Mellon University. 34

[56] Karthikeyan, K. and Indra, A. (2010). Intrusion Detection Tools and Techniques – A Survey. *International Journal of Computer Theory and Engineering*, 2(6). 3, 33

[57] Kent, S. and Atkinson, R. (1998). Security architecture for the internet protocol. Internet RFC 2401. 31

[58] Ku, W. and Chi, C. H. (2004). Survey on the technological aspects of digital rights management. In Zhang, K. and Zheng, Y., editors, *ISC*, volume 3225 of *Lecture Notes in Computer Science*, pages 391–403. Springer. 3, 6

[59] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. (2000). Oceanstore: An architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201. 92

[60] Lampson, B. W. (1971). Protection. In *In Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems Princeton University*, pages 437–443. 42, 43, 47

[61] Lampson, B. W. (1973). A note on the confinement problem. *Commun. ACM*, 16(10):613–615. 49

[62] Lee, A. (2012). The history of file-sharing. 18

[63] Li, P., Mao, Y., and Zdancewic, S. (2003). Information integrity policies. In *Proceedings of The Workshop on Formal Aspects in Security and Trust (FAST)*. 49

[64] Liu, Q., Safavi-naini, R., and Sheppard, N. P. (2003). Digital rights management for content distribution. x, 45, 46

[65] Mazurek, M. L., Arsenault, J. P., Bresee, J., Gupta, N., Ion, I., Johns, C., Lee, D., Liang, Y., Olsen, J., Salmon, B., Shay, R., Vaniea, K., Bauer, L., Cranor, L. F., Ganger, G. R., and Reiter, M. K. (2010). Access control for home data sharing: Attitudes, needs and practices. In *CHI 2010: Conference on Human Factors in Computing Systems*, CHI '10, pages 645–654, New York, NY, USA. ACM. 23, 24, 27

[66] Mclean, J. (1990). Security models and information flow. In *In Proc. IEEE Symposium on Security and Privacy*, pages 180–187. IEEE Computer Society Press. 49

[67] Mendelsohn, J. and Mckenna, J. (2010). Social sharing research report: How, why, and what content people share online. 1

[68] Michelle, K. and Kowalski, E. (2005). Insider Threat Study: Computer System Sabotage in Critical Infrastructure Sectors. 34

[69] Michiels, S., Joosen, W., Truyen, E., and Verslype, K. (2005). Digital rights management - a survey of existing technologies. CW Reports CW428, Department of Computer Science, K.U.Leuven. 6

[70] Microsoft (2016). Microsoft smb protocol and cifs protocol overview. `https://msdn.microsoft.com/en-gb/library/windows/desktop/aa365233(v=vs.85).aspx`. Accessed: 2016-6-1. 92

[71] Miller, A. D. and Edwards, W. K. (2007). Give and take: a study of consumer photo-sharing culture and practice. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 347–356, New York, NY, USA. ACM. 15, 16

[72] Muthitacharoen, A., Morris, R., Gil, T. M., and Chen, B. (2002). Ivy: A read/write peer-to-peer file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):31–44. 92

[73] Myers, A. C. (1999a). Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA. ACM. 50

[74] Myers, A. C. (1999b). Mostly-static decentralized information flow control. Technical report. 50, 51, 161

[75] Myers, A. C. and Liskov, B. (1997). A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31(5):129–142. 50, 161

[76] Myers, A. C. and Liskov, B. (1998). Complete, safe information flow with decentralized labels. In *IEEE Symposium on Security and Privacy*, pages 186–197. IEEE Computer Society. 50, 161

[77] Myers, A. C. and Liskov, B. (2000). Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442. 50, 161

[78] Niemi, A. (2003). End-to-end web security protocols overview. 31

[79] Nistor, C. (2009). File sharing - history. 17, 18

[80] Northcutt, S., Zeltser, L., Winters, S., Kent, K., and Ritchey, R. W. (2005). *Inside Network Perimeter Security (2nd Edition) (Inside)*. Sams, Indianapolis, IN, USA. 32, 33

[81] Olson, J. S., Grudin, J., and Horvitz, E. (2004). Toward understanding preferences for sharing and privacy. MSR Technical Report 2004–138. 19, 20, 23

[82] Olson, J. S., Grudin, J., and Horvitz, E. (2005). A study of preferences for sharing and privacy. In *Proceedings of CHI 05*, pages 1985–1988. ACM Press. 19, 20, 23, 27

[83] Park, J. and Sandhu, R. (2002a). Originator control in usage control. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, POLICY '02, pages 60–, Washington, DC, USA. IEEE Computer Society. 46

[84] Park, J. and Sandhu, R. (2002b). Towards usage control models: Beyond traditional access control. In *In Proceedings of 7th ACM Symposium on Access Control Models and Technologies*. 46

[85] Park, J. and Sandhu, R. (2004). The uconabc usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174. 46

[86] PATRICIU, V.-V., BICA, I., TOGAN, M., and GHITA, S.-V. (2011). A generalized drm architectural framework. *Advances in Electrical and Computer Engineering*, 11:43–48. 45

[87] Paxson, V. (2013). Principles for building secure systems. University of California, Berkeley. Lecture notes. 5

[88] Predd, J., Pfleeger, S. L., Hunker, J., and Bulford, C. (2008). Insiders behaving badly. *IEEE Security & Privacy*, 6(4):66–70. 34, 35

[89] Probst, C. W., Hunker, J., Bishop, M., and Gollmann, D. (2008). 08302 summary – countering insider threats. In Bishop, M., Gollmann, D., Hunke, J., and Probst, C. W., editors, *Countering Insider Threats*, number 08302 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. 34, 35, 36, 59

[90] Probst, C. W., Hunker, J. A., Gollmann, D., and .M, B. (2010). *Insider threats in cyber security*. Advances in Information Security, 49. Springer US. 35, 36, 59

[91] Purohit, V. (2007). Authentication and access control the cornerstone of information security. 41, 42, 43

[92] Qing-hai, B. and Ying, Z. (2011). Study on the access control model in information security. *IEEE*, pages 830–834. 39, 42, 43

[93] Ramsdell, B. and Turner, S. (2004). Secure/multipurpose internet mail extensions (s/mime) version 3.1 message specification", rfc 3851. 31

[94] Rump, N. (2003). Digital rights management - technological, economic, legal and political aspects. In Becker, E., Buhse, W., Gnnewig, D., and Rump, N., editors, *Digital Rights Management*, volume 2770 of *Lecture Notes in Computer Science*. Springer. 5, 6

[95] Sabelfeld, A. and Myers, A. C. (2003). Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19. 48, 49

[96] Sabelfeld, A. and Sands, D. (2009). Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5):517–548. 49

[97] Safavi-Naini, R. and Sheppard, N. P. (n.d). Digital rights management. 45

[98] Salim, F., Reid, J. F., and Dawson, E. (2010). Towards authorisation models for secure information sharing : a survey and research agenda. *ISeCure, The ISC International Journal of Information Security*, 2. 42, 43, 44

[99] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. (1985). Design and implementation or the sun network filesystem. 92

[100] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. (1996). Role-based access control models. *Computer*, 29(2):38–47. 47

[101] Sattarova, F. Y. and Kim, T. (2007). IT Security Review: Privacy, Protection, Access Control, Assurance and System Security. *International Journal of Multimedia and Ubiquitous Engineering*, 2(2). 28

[102] Satyanarayanan, M. (2002). The evolution of coda. *ACM Trans. Comput. Syst.*, 20(2):85–124. 92

[103] Scarfone, K. and Mell, P. (2010). The common configuration scoring system (ccss): Metrics for software security configuration vulnerabilities. Technical Report 7502, National Institute of Standards and Technology. 5, 6

[104] Secure Electronic Transaction LLC (1997). SET secure electronic transaction specification — version 1.0. 31

[105] SecuROM, C. P. ([Accessed: 2013-11-28]). http://www.encrypt.ro/cd-encryption/cd-protection-securom.html. 5

[106] Shirey, R. W. (2000). Internet Security Glossary. Internet RFC 2828. 29

[107] Silowash, G., Cappelli, D., Moore, A., Trzeciak, R., Shimeall, T. J., and Flynn, L. (2012). Common Sense Guide to Mitigating Insider Threats 4th Edition. 37, 59, 60

[108] Simpson, W. A. (1994). The point-to-point protocol (PPP). Internet RFC 1661. 31

[109] Sinclair, S. and Smith, S. W. (2008). Preventative Direction For Insider Threat Mitigation Via Access Control. In Stolfo, S. J., Bellovin, S. M., Keromytis, A., Hershkop, S., Smith, S. W., and Sinclair, S., editors, *Insider Attack and Cyber Security - Beyond the Hacker*, volume 39 of *Advances in Information Security*. Springer. 37, 38

[110] Smetters, D. K. and Good, N. (2009). How users use access control. SOUPS '09. ACM. 22, 23, 24

[111] Smith, M. S. (2011). The history of file sharing: Where did it begin? 17, 18

[112] Spiridonov, D. (2006). Digital rights management. 46

[113] Stallings, W. (2011). *Network Security Essentials - Applications and Standards (4. ed, internat. ed.)*. Pearson Education. 32, 33

[114] Stamp, M. (2003). Digital Rights Management: The Technology Behind the Hype. *Journal of Electronic Commerce Research*, 4(3):102–112. 6

[115] Stamp, M. (2005). *Information security - principles and practice*. Wiley. 6, 7

[116] Stamp, M. (2006). *Information security - principles and practice*. Wiley. 32, 33

[117] Stewart, J. M., Tittel, E., and Chapple, M. (2008). *CISSP: Certified Information Systems Security Professional Study Guide*. SYBEX Inc., Alameda, CA, USA, 4th edition. 40, 41, 42, 43

[118] Sun, S.-T. and Beznosov, K. (2009). Open problems in web 2.0 user content sharing. 1

[119] Takeuchi, K., Honda, K., and Kubo, M. (1994). An interaction-based language and its typing system. In *In PARLE?94, volume 817 of LNCS*, pages 398–413. Springer-Verlag. 54, 55

[120] Vacca, J. (2010). *Network and System Security*. Elsevier Science. 3

[121] Vasconcelos, V. T. (2009). *9th International School on Formal Methods for the Design of Computer, Communication and Software Systems*, volume 5569 of *LNCS*, chapter Fundamentals of Session Types, pages 158–186. SPRINGER. 54

[122] Vaughan, J. A. and Zdancewic, S. (2007). A cryptographic decentralized label model. *2014 IEEE Symposium on Security and Privacy*, 0:192–206. 52

[123] Voida, A., Grinter, R. E., Ducheneaut, N., Edwards, W. K., and Newman, M. W. (2005). Listening in: practices surrounding itunes music sharing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '05, pages 191–200, New York, NY, USA. ACM. 15, 16

[124] Voida, S., Edwards, W. K., Newman, M. W., Grinter, R. E., and Ducheneaut, N. (2006). Share and share alike: Exploring the user interface affordances of file sharing. In *In Proc. of CHI 2006 (April 2227*, pages 221–230. ACM Press. 15, 19, 21, 22, 23, 24, 27

[125] Volpano, D., Irvine, C., and Smith, G. (1996). A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187. 48, 49, 50

[126] Volpano, D. M. and Smith, G. (1997). A Type-Based Approach to Program Security. In *TAPSOFT*, pages 607–621. 48

[127] Wang, J. (2009). *Computer Network Security - Theory and Practice*. Springer London, Limited. 31, 32

[128] Weeks, S. (2001). Understanding trust management systems. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pages 94–, Washington, DC, USA. IEEE Computer Society. 44

[129] Whalen, T., Smetters, D., and Churchill, E. F. (2006). User experiences with sharing and access control. In *In CHI 06: CHI 06 extended abstracts on Human factors in computing systems*, pages 1517–1522. ACM Press. 21, 22, 23, 24, 27

[130] Whalen, T., Toms, E., and Blustein, J. (2008a). File sharing and group information management. Workshop on Personal Information Management (PIM 2008). 15, 16, 17, 21, 22, 23, 25, 27

[131] Whalen, T., Toms, E. G., and Blustein, J. (2008b). Information displays for managing shared files. In *Proceedings of the 2nd ACM Symposium on Computer Human Interaction for Management of Information Technology*, CHiMiT '08, pages 5:1–5:10, New York, NY, USA. ACM. 23, 24

[132] Whitman, M. and Mattord, H. (2011). *Principles of Information Security*. Course Technology Ptr. 28

[133] Wikipedia ([Accessed: 2013-11-28]). Edward snowden. `http://en.wikipedia.org/wiki/Edward_Snowden`. 6

[134] Wikipedia (Accessed on [20/8/2013]b). Timeline of file sharing. 17, 18

[135] Wikipedia (Accessed on [22/8/2013]a). File sharing. 18

[136] windows media player DRM, M. ([Accessed: 2013-11-28]). `http://windows.microsoft.com/en-gb/windows-vista/windows-media-player-drm-frequently-asked-questions`. 5

[137] Yousendit (Accessed on [23/8/2013]). The history of file sharing. 18

[138] Zhang, X. (2011). A survey of digital rights management technologies. `http://www.cs.wustl.edu/~jain/cse571-11/ftp/drm/index.html`. [Accessed: 2013-11-28]. 6, 7

[139] Zheng, L. and Myers, A. C. (2007). Dynamic security labels and static information flow control. *Int. J. Inf. Sec.*, 6(2-3):67–84. 48

[140] Zimmermann, P. R. (1995). *The Official PGP User's Guide*. MIT Press, Cambridge, MA, USA. 31