Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 332 (2017) 3–20

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Controlling File Access with Types

Rakan Alsowail and Ian Mackie

Department of Informatics
University of Sussex
Falmer, UK

Abstract

Accidental misuse of shared files by authorised users is a predominant problem. This paper proposes a well-known static analysis approach, namely a type system, to prevent such accidental misuse. We develop a type system that intercepts commands issued by users in a file system and enforces policies on each file. Commands issued by users to manipulate files will be subject to type checking by the type system. Type-checked commands are then guaranteed to not violate policies of the files. The focus of this paper is on a particular policy that allows owners of files (users who created files) to specify the number of times a file can be read by limiting the number of times a file can be 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. This approach can be extended to other properties.

Keywords: File sharing, security types, type checking

1 Introduction

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 is with insider threats, in particular trusted authorised users who might accidentally violate files policies. The most widely used technique to protect shared files is access control such as Discretionary Access Control (DAC) [9,7] and Role-based Access Control (RBAC) [14]. 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 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.

http://dx.doi.org/10.1016/j.entcs.2017.04.002

Information flow control can be enforced statically [4,5,15,16] or dynamically [13,2]. The former analyses information flow within a program prior to executing, while the latter analyses information during the execution. The dominant approach for enforcing secure information flow statically is the use of type systems.

This paper presents a novel approach of using a type system to solve the problem of accidental misuse of shared files by trusted authorised users. Such misuse occurs, for example, when a trusted authorised user accidentally disseminate a file to unauthorised users, write to a file that is meant be read only, or copy a file that is meant to be read once, after which the file should be erased. Hence, misuse is action that violates files policies. 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 paper, we focus on enforcing a particular policy, namely, limiting the number of times a file can be read. However, the same basic ideas can be extended to enforce other policies as pointed out in Section 6.

The rest of this paper is organised as follows. Section 2 presents the security types and policies of files. Section 3 describes the language syntax and semantics for manipulating files, defines security errors and an algorithm for checking syntactical errors. Section 4 describes the type system, and includes properties. Section 5 introduces a type checking algorithm and proves its soundness and completeness. In Section 6 we give a brief review of related work, and finally we conclude the paper in Section 7.

2 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, a file can be read as much as it can be copied. If a file cannot be copied, then it can be read once. To enforce this policy, we need to restrict the access to *copy* operations and restrict the information flow caused by all operations such that restrictions of files over *copy* operations are not violated. 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. 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.

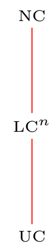

Fig. 1. Security types

To control the information flow among files, our security copy types form a lattice $(\tau, \sqsubseteq)$, where $\tau = \{\mathrm{NC}, \mathrm{LC}^n, \mathrm{UC}\}$, are partially ordered by $\sqsubseteq$ (see Figure 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 $\mathrm{LC}^n \sqsubseteq \mathrm{LC}^{n'}$ if and only if $n \succeq n'$. To formally state the policies we need to enforce, we define the following functions and notations. 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. That is, $dst(\mathrm{UC}) = \mathrm{UC}$ and $dst(\mathrm{LC}^n) = \mathrm{NC} \iff n > 0$. 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. That is, $red(\mathrm{UC}) = \mathrm{UC}$ and $red(\mathrm{LC}^n) = \mathrm{LC}^{n-1} \iff n > 0$. Note that we do not define $dst(\mathrm{LC}^{n\leq 0})$ nor $dst(\mathrm{NC})$ or $red(\mathrm{LC}^{n\leq 0})$ and $red(\mathrm{NC})$, because files of these types are not allowed to be copied, and hence, applying the functions on them should result in an error. Let $T(f)$ denotes the type associated with the file f, and $T(f_1) \sqcup T(f_2)$ denotes the least upper bound of the types of f_1 and f_2. That is, if $T(f_1) = \mathrm{NC}$ and $T(f_2) = \mathrm{UC}$, then $T(f_1) \sqcup T(f_2) = \mathrm{NC}$. Let $f_1 \rightarrow^{copy} f_2$ denotes a flow of information from f_1 to f_2 caused by copy operation, $f_1 \rightarrow^{o} f_2$ denotes a flow of information caused by other operations than copy, such as `mv`, `cat`, etc., and $f_1 \in types$ denotes f_1 is associated with a security type. Below we give the definitions of the policies to be enforced by our type system.

Definition 2.1 $\forall f_1, f_2 \in types.\ f_1 \rightarrow^{o} f_2$ is always allowed, provided that f_2 must change its type to $T(f_1) \sqcup T(f_2)$ and f_1 must be consumed after performing the operation.

Definition 2.2 $\forall f_1, f_2, f_3 \in types.\ f_1, f_2 \rightarrow^{o} f_3$ is always allowed, provided that f_3 must change its type to $T(f_1) \sqcup T(f_2) \sqcup T(f_3)$ and f_1 and f_2 must be consumed after performing the operation.

Definition 2.3 $\forall f_1, f_2 \in \mathit{types}.\ f_1 \rightarrow^{copy} f_2$ 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.

It should be noted that when a file is consumed, the file must not be available for any subsequent operations (i.e.the file must be erased). Next we present our language syntax and semantics, define the notion of security errors, and an algorithm for checking syntactical errors.

3 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:

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

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, $\delta = \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2, \ldots, f_n(c_n) : \tau_n\}$. In later sections we might refer to δ as the set of file names only (e.g., $\delta = \{f_1, f_2, \ldots, f_n\}$) or the set of file names with types only (e.g., $\delta = \{f_1 : \tau_1, f_2 : \tau_2, \ldots, f_n : \tau_n\}$). It should be apparent from the context which δ we mean. 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. 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. Finally, we write $\langle c, \delta \rangle \rightarrow \delta'$ for evaluating the command c in δ that yields a new file system δ'. For example, if $\delta = \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2\}$, then $\langle \texttt{rm}\ f_1, \delta \rangle \rightarrow \{f_2(c_2) : \tau_2\}$, $\langle \texttt{mv}\ f_1\ f_2, \delta \rangle \rightarrow \{f_2(c_1) : T(f_1) \sqcup T(f_2)\}$, and $\langle \texttt{append}\ f_1\ f_2\ f_3, \delta \rangle \rightarrow \{f_3(C(f_1) + C(f_2)) : T(f_1) \sqcup T(f_2)\}$. If any of the constraints of the operations applied to δ are not satisfied, then evaluating the configuration $\langle c, \delta \rangle$ should lead to an error, written as $\langle c, \delta \rangle \rightarrow \mathit{Err}$. 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. These are given below.

$$\langle \texttt{cp}\ f_1\ f_2\ , \delta\ \rangle \rightarrow \delta[f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_2) \sqcup \mathit{dst}(T(f_1))][f_1 \leftarrow \mathit{red}(T(f_1))]$$

$$\langle \texttt{rm}\ f, \delta\rangle \rightarrow \delta[-f]$$

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

$$\langle \texttt{rd}\ f, \delta\rangle \rightarrow \delta[-f]$$

$$\langle \texttt{cat}\ f_1\ f_2\ f_3, \delta\rangle \rightarrow$$

$$\delta[f_3 \leftarrow C(f_1) + C(f_2)][f_3 \leftarrow T(f_1) \sqcup T(f_2) \sqcup T(f_3)][-f_1, -f_2]$$

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

From the single-step transitions, we can define the semantics of sequences of commands as the (small-step) operational semantics as shown below:

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

We write $\Rightarrow^*$ as usual for the reflexive and transitive closure of $\Rightarrow$. Therefore, a sequence of commands can be computed by: $\langle c, \delta\rangle \Rightarrow^* \delta'$.

3.1 Security Errors

Security errors can be divided into syntactical errors and type errors. Syntactical errors occur when the constraints of an operation applied to δ are not satisfied. For example, if $\delta = \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2\}$, then evaluating the configuration $\langle \texttt{rm}\ f_4, \delta\rangle$ should lead to an error, that is $\langle \texttt{rm}\ f_4, \delta\rangle \rightarrow \mathit{Err}$, because the operation $\delta[-f_4]$ requires f_4 to exist in δ. On the other hand, type errors occur when the constraints a function applied to a type of a file are not satisfied. For example, if $T(f_1) = \text{NC}$, then $\langle \texttt{cp}\ f_1\ f_2, \delta\rangle \rightarrow \mathit{Err}$, because the function $\mathit{dst}(\text{NC})$ results in an error. Syntactical and type constraints are shown in Table 1 and 2. It can be seen that content update operations in Table 1 (i.e. $\delta[f_2 \leftarrow C(f_1)]$ and $\delta[f_3 \leftarrow C(f_1) + C(f_2)]$) require files names to be distinct. Such constraint is very important to rule out errors that result from evaluating configurations such as $\langle \texttt{cat}\ f_1\ f_1\ f_3, \delta\rangle$, which erases f_1 twice, and to rule out accidental erasure of files that result from evaluating configurations such as $\langle \texttt{cat}\ f_1\ f_2\ f_1, \delta\rangle$ and $\langle \texttt{mv}\ f_1\ f_1, \delta\rangle$, which have the same effect as evaluating the configuration $\langle \texttt{rm}\ f_1, \delta\rangle$. Next we present our algorithm for checking syntactical errors.

3.2 Syntactical checking for correctness

An occurrence of a file name in a command determines whether or not the command can be successfully evaluated in a particular file system δ. Some commands, such as ($\texttt{rm}\ f$), require the file name to exist in δ, while others, such as ($\texttt{mkf}\ f\ t$), require them to not exist. Checking commands individually with respect to a particular

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 1
Constraints of operations applied to δ

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

Table 2
Constraints of operations applied to types

file system does not work however. For example, consider the file system $\delta = \{f_1(c_1) : \tau_1, f_2(c_2) : \tau_2\}$, then the configuration $\langle \texttt{rm}\ f_1; \texttt{rm}\ f_1, \delta\rangle$ will fail to evaluate because the first command will remove f_1 from δ so that the second will generate an error. A similar situation arises with the configuration $\langle \texttt{mkf}\ f_3\ t; \texttt{mkf}\ f_3\ t, \delta\rangle$. In this case, the first command will succeed, but the second will fail because the file f_3 is already there. Therefore, since command evaluation changes δ, such changes must be considered by subsequent commands when checking occurrences of file names. We define an algorithm to check consistency of commands in the following way. For a given command cs, we compute a 4-tuple (H, N, C, E) that gives the constraints on a starting file system δ so that it can execute without 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 the set of file names that are created by the sequence of commands, such file names do not necessarily have to be in δ initially. E denotes the set of file names that are erased by the sequence of commands, so such files do not necessarily have to be free in δ initially. Table 3 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. Sequences of commands are then computed by composition: $c; cs(H, N, C, E) = cs(c(H, N, C, E))$. The algorithm starts with $(\varnothing, \varnothing, \varnothing, \varnothing)$.

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

Table 3
Algorithm for syntactically checking commands

An example of using the algorithm is the command $\texttt{cp}\ f_1\ f_2$:

$$\texttt{cp}\ f_1 f_2\ (\varnothing, \varnothing, \varnothing, \varnothing) = (\{f_1, f_2\}, \varnothing, \varnothing, \varnothing).$$

This means that the files $\{f_1, f_2\}$ must be part of the file system when this command is executed. A second example is a sequence of commands $\texttt{mkf}\ f_1\ t; \texttt{rm}\ f_1$:

$$\begin{aligned}\texttt{mkf}\ f_1\ t;\ \texttt{rm}\ f_1(\varnothing,\varnothing,\varnothing,\varnothing) &= \texttt{rm}\ f_1(\texttt{mkf}\ f_1\ t(\varnothing,\varnothing,\varnothing,\varnothing))\ \textit{Since } f_1 \notin C, \textit{then}\\ &= \texttt{rm}\ f_1(\varnothing,\{f_1\},\{f_1\},\varnothing)\ \textit{and since } f_1 \notin E, \textit{then}\\ &= (\varnothing,\{f_1\},\varnothing,\{f_1\})\end{aligned}$$

This means that there are no files needed in δ to execute these commands successfully, and if there are any files, then the file name f_1 cannot be used. When a command does not satisfy the conditions in the table, the algorithm fails. Consider a sequence of commands $\texttt{mkf}\ f_1\ t; \texttt{mkf}\ f_1\ t$:

$$\begin{aligned}\texttt{mkf}\ f_1\ t;\ \texttt{mkf}\ f_1\ t(\varnothing,\varnothing,\varnothing,\varnothing) &= \texttt{mkf}\ f_1\ t(\texttt{mkf}\ f_1\ t(\varnothing,\varnothing,\varnothing,\varnothing))\ \textit{Since } f_1 \notin C\\ &= \texttt{mkf}\ f_1\ t(\varnothing,\{f_1\},\{f_1\},\varnothing)\end{aligned}$$

however, since the condition $f_1 \notin C$ is not satisfied, then $\texttt{mkf}\ f_1\ t(\varnothing,\{f_1\},\{f_1\},\varnothing)$ fails. Similarly, a sequence of commands such as $\texttt{rm}\ f_1; \texttt{rm}\ f_1$:

$$\begin{aligned}\texttt{rm}\ f_1;\ \texttt{rm}\ f_1(\varnothing,\varnothing,\varnothing,\varnothing)) &= \texttt{rm}\ f_1(\texttt{rm}\ f_1(\varnothing,\varnothing,\varnothing,\varnothing)).\ \textit{Since } f_1 \notin E,\ \textit{then}\\ &= \texttt{rm}\ f_1(\{f_1\},\varnothing,\varnothing,\{f_1\})\end{aligned}$$

however, since the condition $f_1 \notin E$ is not satisfied, then, $\texttt{rm}\ f_1(\{f_1\},\varnothing,\varnothing,\{f_1\})$ should fail. Table 3 captures the idea that any file name of a command that needs to be in δ must not have been removed by previous commands and any file name of a command that needs not to be in δ must not have been created by previous commands. Furthermore, file names of a command must be distinct. 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 execute successfully (i.e., without error). Essentially, this key result gives the constraints on the file system: which files must be present, and which files must not be present.

Theorem 3.1 *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 there exists a file system* δ' *such that* $\langle cs,\delta\rangle \Rightarrow^* \delta'$.

4 Type System

Now we present the type system that will check commands to ensure they are free of syntactical and type errors before execution. Typing judgements have the form $\Gamma \mid \Gamma' \vdash p : \tau$ where Γ is a list of files with types of the form $f : \tau$. We write $\varnothing$ for the empty list. 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 list 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 or a file name. The typing rules are shown in Figure 2. Rule (f) says that typing a file from the context Γ consumes the file from the context. Rule (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''. Rule (`cp`) 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 $dst(f_2)$ and the type of f_1 is changed to be the type of $red(f_1)$. Rules (`rm`) and (`rd`) say that if we can type f from the context Γ, then we can type the command `rm` f of type void and the command `rd` f of type void, respectively. Rule (`mkf`) says that typing the command `mkf` f t of type void will add f of type t to the context Γ. Rule (`cat`) 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. Rule (`mv`) 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 and 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. Examples of commands derivations are given in Appendix A.

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

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

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

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

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

Fig. 2. Typing rules

4.1 Properties of Type System

We prove the soundness and completeness of our type system with respect to the operational semantics. The soundness property is proved by showing two properties which are preservation and 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 c is typable in a particular file system δ, then the command c must take a step of evaluation.

Theorem 4.1 (Progress) *If* $\Gamma = \delta$ *and* $\Gamma \mid \Gamma' \vdash c : \tau$*, then* $\langle c, \delta\rangle \not\rightarrow \mathit{Err}$.

Proof. We proceed by cases on typing derivation of c. There are 6 cases, we show only two.

(i) $c = \texttt{cp}\ f_1\ f_2$
We know there is a typing derivation for c 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 $\Gamma' \mid \Gamma'' \vdash f_2 : \tau'$. Now we can use rule (1) to obtain $\langle \texttt{cp}\ f_1\ f_2\ , \delta\rangle \rightarrow \delta[f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_2) \sqcup \mathit{dst}(T(f_1))][f_1 \leftarrow \mathit{red}(T(f_1))]$. Since the configuration $\langle \texttt{cp}\ f_1\ f_2\ , \delta\rangle$ 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, $\langle \texttt{cp}\ f_1\ f_2\ , \delta\ \rangle \not\rightarrow^s \mathit{Err}$. 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, $\langle \texttt{cp}\ f_1\ f_2\ , \delta\ \rangle \not\rightarrow \mathit{Err}$ as required.

(ii) $c = \texttt{rd}\ f$
We know there is a typing derivation for c by using rule ($\texttt{rd}$) with conclusion: $\Gamma \mid \Gamma' \vdash \texttt{rd}\ f : \text{void}$. We must also have a sub-derivation 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 $\langle \texttt{rd}\ f, \delta\rangle \not\rightarrow \mathit{Err}$.

□

Traditionally, the preservation theorem states that as we evaluated a program, its type is preserved at each evaluation step. 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 4.2 (Preservation) *If* $\Gamma = \delta$ *and* $\Gamma \mid \Gamma' \vdash c : \tau$*, and* $\langle c, \delta\rangle \rightarrow \delta'$*, then* $\Gamma' = \delta'$.

Proof. We proceed by cases on $\langle c, \delta\rangle \rightarrow \delta'$. There are 6 cases, we only show two.

(i) $c = \langle \texttt{cp}\ f_1\ f_2\ , \delta\ \rangle \rightarrow \delta[f_2 \leftarrow C(f_1)][f_2 \leftarrow T(f_2) \sqcup \mathit{dst}(T(f_1))][f_1 \leftarrow \mathit{red}(T(f_1))]$
We know there is a typing derivation for c 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 6 cases based on typing f_1 and f_2, we show two of

them.

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

In this case, the typing derivation of c must have the form $\Gamma \mid \Gamma'', f_1 : \mathrm{LC}^{n-1},\ f_2 : \mathrm{NC} \vdash \texttt{cp}\ f_1\ f_2 : \mathrm{void}$. Let $\Gamma' = \Gamma'', f_1 : \mathrm{LC}^{n-1},\ f_2 : \mathrm{NC}$. 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 $\mathrm{LC}^{n>0} \neq \mathrm{LC}^{n-1}$ and $\mathrm{UC} \neq \mathrm{NC}$, 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 $\mathrm{LC}^{n>0} \neq \mathrm{LC}^{n-1}$ and $\mathrm{UC} \neq \mathrm{NC}$, 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 $\mathrm{LC}^{n-1} = \mathrm{LC}^{n-1}$ and $\Gamma'(f_2) = \delta'(f_2)$ that is $\mathrm{NC} = \mathrm{NC}$, and $\Gamma = \delta$. Then, $\Gamma' = \delta'$ as required.

(b) : $\Gamma \mid \Gamma' \vdash f_1 : \mathrm{LC}^{n>0}, \Gamma' \mid \Gamma'' \vdash f_2 : \mathrm{NC}$

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

(ii) $c = \langle \texttt{rm}\ f, \delta \rangle \to \delta[-f]$

We know there is a typing derivation for c by using rule ($\texttt{rm}$) with conclusion: $\Gamma \mid \Gamma' \vdash \texttt{rm}\ f : \mathrm{void}$. We must also have sub-derivation 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.

□

Now we show the completeness property of the type system with respect to the operations semantics. The completeness property is useful since it shows that the operational semantics are not unnecessarily restricted by the type system. That is the type system does not provide false negative results. The completeness theorem states that if a command c can be evaluated in a particular file system δ without an error, then the command c must be typable.

Theorem 4.3 (Completeness) *If $\langle c, \delta \rangle \not\to Err$ and $\Gamma = \delta$, then $\Gamma \mid \Gamma' \vdash c : \tau$ for some Γ'.*

Proof. We proceed by cases on commands c. There are 6 cases, we show only two.

(i) $c = \texttt{cp}\ f_1\ f_2$

If $\langle \texttt{cp}\ f_1\ f_2, \delta \rangle \not\to Err$, then it must be the case that $f_1 \neq f_2$, $f_1 \wedge f_2 \in \delta$ and $\delta(f_1) \in \{\mathrm{UC}, \mathrm{LC}^{n>0}\}$. Therefore, since $\Gamma = \delta$, we know there is a typing derivation for f_1 and f_2, and $\Gamma(f_1) \in \{\mathrm{UC}, \mathrm{LC}^{n>0}\}$:

$$\frac{}{\Gamma \mid \Gamma - \{f_1\} \vdash f_1 : \tau}\,(f) \qquad \frac{}{\Gamma \mid \Gamma - \{f_2\} \vdash f_2 : \tau'}\,(f)$$

Now by the (`cp`) rule, there is also a derivation:

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

(ii) $c = \texttt{rd}\ f$

If $\langle \texttt{rd}\ f, \delta \rangle \not\to Err$, then it must be the case that $f \in \delta$. Therefore, since $\Gamma = \delta$, we know there is a typing derivation for f:

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

Now by the (`rd`) rule, there is also a derivation:

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

□

It is useful to show that commands in our language are monotonically increasing, in the sense that types of files never decrease during commands evaluation. This is a straightforward property since we only allow commands to change a type of a file to be the least upper bound of its type and the source type. Since the least upper bound of any two types will be at least as restrictive as both of them, types of files will never decrease.

Theorem 4.4 (Monotonicity of files types) *If* $\Gamma \mid \Gamma' \vdash c : \tau$, *then* $\forall f, f' \in \Gamma \wedge \Gamma'$, *if* $\Gamma(f) \sqsubseteq \Gamma(f')$, *then* $\Gamma'(f) \sqsubseteq \Gamma'(f')$

5 Typing Algorithm

Here we give an algorithm $\mathcal{T}$ for typing commands. We define a number of helper functions: $\mathit{check}(\alpha, \beta)$ returns true if the types are compatible. Note that any two different base types are not compatible, e.g., $\mathit{check}(LC^n, \mathrm{void})$ will fail. $\mathit{less}(\tau, \tau')$ returns true if $\tau \sqsubseteq \tau'$. $\mathit{lub}(\tau, \ldots, \tau_n)$ returns the least upper bound of all its parameters i.e., $\tau \sqcup \ldots \sqcup \tau_n$. Using these functions, we can now define the type checking algorithm $\mathcal{T}$:

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

where:

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

(ii) If e is a sequence of commands, $c; cs$ let

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

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

(iii) If e is the command `cp` f_1 f_2 let

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

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

(iv) If e is the command `rm` f let $(\alpha, A_1) = \mathcal{T}(A, f)$, then $\tau = \text{void}, A' = A_1$.

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

(vi) If e is the command `rd` f let $(\alpha, A_1) = \mathcal{T}(A, f)$, then $\tau = \text{void}, A' = A_1$.

(vii) If e is the command `cat` f_1 f_2 f_3 let

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

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

(viii) If e is the command `mv` f_1 f_2 let

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

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

5.1 Properties of Algorithm $\mathcal{T}$

If $\mathcal{T}$ finds a type for a command, then there is a derivation for that command. This property is called soundness, and means that the algorithm will not give wrong answers. We first prove soundness, then show that the algorithm can find all derivations, a property called completeness.

Theorem 5.1 (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 command e. There are 8 cases, here we show a selection of them.

(i) 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.

(ii) 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.

(iii) If e is the command $\texttt{cp}\ f_1\ f_2$, 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 $\mathit{less}(\beta, LC)$, we have $\beta \sqsubseteq LC$ and now we can use the $(\texttt{cp})$ rule to give a derivation of $A \mid A_2, f_1 : \beta, f_2 : \alpha \sqcup \mathit{dst}(\beta) \vdash \texttt{cp}\ f_1\ f_2 : \text{void}$ as required.

(iv) If e is the command $\texttt{rm}\ f$, 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 $(\texttt{rm})$ rule to give a derivation of $A \mid A_1 \vdash \texttt{rm}\ f : \text{void}$ as required.

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

□

Theorem 5.2 (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 command e. There are 8 cases, here we show a selection of them.

(i) 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.

(ii) If e is the sequence of commands $c; cs$, then by (cs) rule there is a derivation ending in $A \mid A_2 \vdash c; cs : \mathit{void}$ 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).

(iii) If e is the command $\texttt{cp}\ f_1\ f_2$, then by $(\texttt{cp})$ rule, there is a derivation ending in $A \mid A_2, f_1 : \beta, f_2 : \alpha \sqcup \mathit{dst}(\beta) \vdash \texttt{cp}\ f_1\ f_2 : \text{void}$ which consists of two derivations: $A \mid A'_1 \vdash f_1 : \beta$ and $A \mid A_2 \vdash f_2 : \alpha$. By the induction hypothesis twice and since $\beta \sqsubseteq LC$ we have $\mathit{less}(\beta, LC)$, $\mathcal{T}(A, f_1)$ 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 $\mathcal{T}(A, (\texttt{cp}\ f_1\ f_2))$ succeeds with $(\text{void}, A_2 \cup \{f_1 : \beta, f_2 : \mathit{lub}(\alpha, \mathit{dst}(\beta))\})$.

(iv) If e is the 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 $\mathcal{T}(A, (\texttt{rm}\ f))$ succeeds with (void, A_1).

(v) 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, $\mathcal{T}(A, (\texttt{mkf}\ f\ t))$ succeeds with $(\text{void}, A \cup \{f : \tau\})$ as required.

□

6 Related Work and Discussion

Denning [5] 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 [1,8,17] beginning with Volpano et al. [15] and Volpano and Smith [16] who were the first to formulate Denning's secure information flow analysis [4,5] 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. A comprehensive survey of the large body of work on this aspect, can be found in [12].

The majority of security type systems focus on enforcing a property known as *non-interference* [6,15,12,11]. 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. However, non-interference is a very restrictive property since it does not allow downgrading of security levels from high to low which in practice is needed in many applications. We take a different approach to non-interference, and define our security property as the absence of run-time errors which are raised by commands that violate files policies. In this way we could easily augment our language with commands that declassify the security levels of information and allow owners of files to execute such commands. Hence, executing such commands by a user who is not an owner should lead to an error.

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 [10,3]. 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. This is because conventional security levels represent only information flow policies which restrict the information flow but not the operations which cause such flow. Our security types, however, represent both access control and information flow policies which restrict the operations to be performed on types and the information flow caused by performing the operations.

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 [15], 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. On the other hand, in flow-sensitive type systems [8], 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 information flow policy enforced by our type system is somewhere in between the flow policies enforced by flow-insensitive and flow-sensitive type systems. We follow the idea of flow-insensitive type system in that flow of information must only result in more restrictive type of information while we follow the idea of flow-sensitive type system in that information can flow any where and the security types can be changed during computation. This can be achieved by allowing information to flow from a security 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 separately. In this way we benefit from the flexibility flow-sensitive type systems of and the restrictiveness of flow-insensitive type systems.

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.

The type system developed in this paper is only concerned with *copy* operations. This is because it enforces the number of times a file can be read by limiting the number of copies a file can produce. Therefore, the defined security types in this paper specify policies that restrict the access to *copy* operations (access control policy), and restrict the information flow caused by all operations including *copy*, such that copy policies of files are not violated (information flow policy). Other policies can be enforced similarly by defining new security types that restrict the access to the relevant operations and the flow caused by them. For example, to specify policies that restrict the access to read and write operations which are caused by `rd`, `cat`, `cp` and `mv` operations, we can define the following security types $\mathrm{NRW}, \mathrm{RO}, \mathrm{WO}^-, \mathrm{WO}^+, \mathrm{RW}^-, \mathrm{RW}^+$. NRW stands for NoReadOrwrite, RO stands for ReadOnly, WO stands for WriteOnly, and RW stands for ReadWrite. The symbol $(+)$ means a file of this type cannot be overwritten while the symbol $(-)$ means a file of this type can be overwritten. Since we can only write to a file of type RW or WO, these symbols are associated only with them. Hence the typing rule should restrict `rd` operation to be performed on a file of type $\sqsubseteq RO$, `cat` operation to be performed on source files of type $\sqsubseteq WO^+$ and a destination file of type $\sqsubseteq WO^-$, `cp` operation to be performed on destination file of type $\sqsubseteq WO^-$, and `mv` operation to be performed on destination file of type $\sqsubseteq WO^-$. To restrict the flow of information caused by these operations such that policies for read and write operations are not violated, we let these types form a lattice of security types $(\tau, \sqsubseteq)$ where $\tau = \{\mathrm{NRW}, \mathrm{RO}, \mathrm{WO}^-, \mathrm{WO}^+, \mathrm{RW}^-, \mathrm{RW}^+\}$, are partially ordered by $\sqsubseteq$ (see Figure 3). Hence, information can only flow from less restrictive types to more restrictive types.

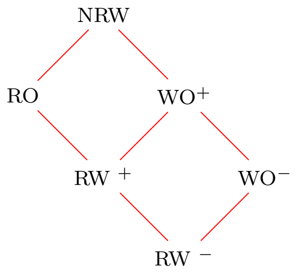

Fig. 3. Security access types

7 Conclusion

In this paper we have presented our approach to prevent accidental misuse of the shared files. The focus was on enforcing a particular policy, namely limiting the number of times a file can be read. This is achieved by developing a type system that controls the access to copy operations and the flow caused by all operations including copy, such that the policies for copying files are not violated. The type system plays the role of a reference monitor that intercepts each command and checks whether or not the command will cause run-time errors. Run-time errors are caused by executing commands that violate files policies. Therefore, type-checked commands are safe to be executed since they do not cause any policy violation. We have proven the soundness and completeness of the type system with respect to the operations semantics and define a type checking algorithm that is shown to be sound and complete.

The language and the type system presented in this paper 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. 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.

So far we have taken a significant step towards realising these features. In particular, we extended the type system to enforce additional policies to control *read* and *write* operations. By defining a new set of security types to control *read* and *write* operations, we found that the same typing rules presented in this paper with a few additional constraints can be used to enforce the new policies. Therefore, we have added these additional constraints to the current type system and 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. The resulting type system controls the access to copy, read and write operations and the flow caused by all operations.

We have also extended the policies of files to specify which operations can be performed on which types of files and by whom. We have done this by defining security types of files as labels that not only consist of a security copy type and a security access type, but also of ownership and authorisation information. The ownership and authorisation information in a label indicates the owners and the authorised users of a file associated with the label. 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. Furthermore, we have also looked at possible extensions to allow owners to manipulate their files policies. Thus, we extended the commands in our language to include commands that manipulate file policies. Accordingly, we extended the type system with typing rules for these commands along with a typing algorithm for typing phrases.

While the extensions described above have been added to our current work, they are still missing an important aspect. Further investigation and formal proofs their properties are required. We are currently developing various proofs of the extended system to ensure the desired properties of the system are still preserved.

References

[1] M. Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, Sept. 1999.

[2] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. *SIGPLAN Not.*, 44(8):20–31, Dec. 2009.

[3] A. Birgisson, A. Russo, and A. Sabelfeld. Unifying facets of information integrity. In S. Jha and A. Mathuria, editors, *ICISS*, volume 6503 of *Lecture Notes in Computer Science*, pages 48–65. Springer, 2010.

[4] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[5] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[6] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

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

[8] S. Hunt and D. Sands. On flow-sensitive security types. *SIGPLAN Not.*, 41(1):79–90, Jan. 2006.

[9] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, Jan. 1974.

[10] P. Li, Y. Mao, and S. Zdancewic. Information integrity policies. In *Proceedings of The Workshop on Formal Aspects in Security and Trust (FAST)*, 2003.

[11] J. Mclean. Security models and information flow. In *In Proc. IEEE Symposium on Security and Privacy*, pages 180–187. IEEE Computer Society Press, 1990.

[12] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[13] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 352–365. Springer, 2009.

[14] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, Feb. 1996.

[15] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, Jan. 1996.

[16] D. M. Volpano and G. Smith. A Type-Based Approach to Program Security. In *TAPSOFT*, pages 607–621, 1997.

[17] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *Int. J. Inf. Sec.*, 6(2-3):67–84, 2007.