# Enforcing Well-formed and Partially-formed Transactions for Unix

Dean Povey
*Security Unit*
*Cooperative Research Centre for Enterprise Distributed Systems*
*Queensland University of Technology, Brisbane, 4001*
povey@dstc.edu.au

## Abstract

While security is a critical component of information systems, at times it can be frustrating for end users. Security systems exist to minimise the risks of allowing users to access and modify data, but rarely do they consider the risks of *not* granting access.

This paper describes an access control system which is *optimistic*, i.e. it assumes accesses are legitimate, and allows audit and recovery of the system when they are not. The concepts of *well-formed* and *partially-formed* transactions as mechanisms for constraining pessimistic and optimistic accesses is briefly described, and the paper details a prototype implementation for the Solaris operating system which provides a reference monitor for enforcement of both these transactions.

## 1 Introduction

One of the main objectives of security systems is the management of risks by controlling access to resources. These systems implement security policies which seek to enforce the principles of *least privilege* and *separation of duties* to ensure that users are given the minimum amount of privilege necessary to perform tasks.

However, what most systems often do not consider is the risk of *not* granting privileges. Either due to unforeseen circumstances, or because the access control system is not flexible enough to support complex policies, enforcing restrictive access control can impede a user's legitimate activities.

This paper considers security from the perspective of reducing risks in an environment where access control is performed optimistically. In such a system, access is assumed to be legitimate, and granted automatically. Integrity is ensured by verifying the validity of actions after the fact, and providing a way of rolling back user actions when they are deemed inappropriate. The paper categorises user actions as being one of three types: legitimate, questionable and dangerous, and shows how both pessimistic and optimistic access control mechanisms are useful for constraining different classes of action.

The notion of *partially-formed transactions* (an extension of Clark and Wilson's *well-formed transaction*[5]) is described as a mechanism for ensuring integrity in an optimistic access control system and this concept is presented within a formal context.

The paper also describes the design and implementation of `tudo`, a prototype for the Unix operating system that enforces both well-formed and partially-formed transactions. `tudo` provides a reference monitor which controls access to system calls using the system call tracing features of the Solaris `/proc` filesystem, and provides detailed logging of actions and recovery from filesystem modifications. The security and limitations of the prototype are evaluated and some future directions for the research are discussed.

## 2 Motivation

In designing access control policies for an organisation, security administrators need to balance the needs of users with the need to maintain system integrity. However, these two requirements are often conflicting. In general, it is most efficient for users to have minimal barriers to completing tasks, particularly in environments which are collaborative and time-critical. Maintaining system integrity however, requires that a user's access privileges be restricted to a minimal set, to ensure that the amount of damage that can be caused (accidental or otherwise) is minimised.

The situation is further exacerbated in environments such as Unix, where the simple access control scheme supported by the system is often not flexible enough to support complex security policies. For example, in Unix, it is not possible to authorise the following actions without giving the user additional privileges:

- allow append-only access to a file;
- bind a network socket to a port $< 1024$;
- allow read/write access to a raw disk device; or
- allow mounting or unmounting of a filesystem.

Furthermore, security systems usually enforce a policy appropriate for the normal operation of a system, but do not allow flexibility in dealing with disaster scenarios or dynamically changing environments.

For example, suppose a user with a critical deadline turns up early to work to find that the line printer daemon configuration has been changed, and they are unable to print their project proposal. As the system administrator is nowhere to be found, their only option is to attempt to fix the problem themselves. In normal circumstances, allowing the user to alter the line printer configuration file would be unacceptable, however in this situation, not allowing them to alter it will mean that they cannot meet their deadline.

## 2.1 Categories of user actions

When considering how a given access control mechanism implements its security policy, we can think of the set of possible actions a user might wish to perform as being divided into three separate categories:

**Legitimate Actions** that are explicitly allowed by a system's access control mechanism.

**Questionable Actions** which may represent legitimate user behaviour, but require access privileges not normally given to the user.

**Dangerous Actions** that are explicitly disallowed by the security mechanism, and which represent accidental damage or malicious intent by users.

Security systems only allow legitimate actions. However, either due to unforeseen circumstances, or because the access control system is not sufficient to support the complexity of authorisations necessary for a particular task, a user will occasionally need to perform actions which will be deemed questionable. A user who needs to perform such actions has two options:

1. convince a benevolent system administrator to perform the action for them; or
2. perform the action through a trusted entry point provided by the administrator.

### 2.1.1 Questionable actions through a system administrator

Most operating systems (including Unix) have the concept of a privileged user or superuser who can override the security mechanisms of the system to perform a questionable action. However, there are often occasions (such as the example above) when the system administrator cannot be found (or a *benevolent* system administrator cannot be found), or where they are too busy to complete the request in a timely fashion. In addition, requiring all questionable actions to be approved by the system administrator is inefficient, particularly when the action is commonly performed.

### 2.1.2 Questionable actions through trusted entry points

Unix supports the notion of trusted entry points, by allowing a program to be executed with setuid or setgid permissions. When executed, the program is run with the privileges of either the owner or group of the executable file (depending on which permission is set). The main benefit of this approach is it allows the administrator to authorise questionable actions which are regularly executed, and known to be trusted. However, in many cases, the privileges required to perform the operation will demand that the program be executed as the superuser. Because executing with the superuser privilege provides full access to the system, these programs then often become a target for misuse by exploitation of weaknesses in their design and implementation (for examples of these exploits see [12] and [15]). The existence of such widespread exploits make administrators reticent to provide too many entry points for users to execute questionable actions, as they do not always have the time or expertise to determine if the setuid/setgid program has been implemented securely.

Another approach is the use of utilities such as `sudo`[19] and `super`[20], that allow the execution of specified programs with superuser privileges. The advantages of such utilities over normal setuid/setgid programs are that they provide richer access control semantics using a high level policy language, and they sanitise the environment in which the specified programs run to avoid known attacks. This simplifies the process of certifying a particular program as being secure, as many vulnerabilities are averted by the `sudo` and `super` utilities themselves.

However, such approaches are still susceptible if the programs they execute behave in unexpected ways or are vulnerable to data dependent attacks. The `sudo` and `super` utilities provide a safer way to access trusted entry points, but once the entry point has been authorised, the program being run has carte blanche access to the system.

Lastly, mechanisms which support trusted entry points are only useful for cases where the need for a questionable action is known in advance. In the example given above, it is unlikely that an administrator will to provide a entry point to allow users to modify the printer configuration file, as such access is probably legitimate only as a one-off occurrence.

## 2.2 Optimistic access control

Another way of approaching the issue of authorising questionable access, is to decide whether authorisation should be pessimistic or optimistic. In the schemes described above, authorisation is pessimistic, i.e. it is assumed that all accesses are potentially dangerous, and only those cases explicitly authorised by the administrator are permitted.

An optimistic approach, is to assume that most accesses users will request will be legitimate, and any access that does not fall in the dangerous category should be granted. The actions are audited so that if a system administrator later decides that an access was unreasonable, they can take corrective action to repair any damage. Because the scheme is optimistic it assumes that such instances will be rare. For example, in the situation above where a user needed to edit the printer configuration file to fix a problem, the system administrator should be able to determine that under the circumstances, such action was legitimate. In the case that a user acted inappropriately in performing such an action, the administrator can take steps to make sure that the user is aware of their mistake, take punitive action, or specifically disallow them that privilege in future.

In an optimistic system, it is the responsibility of users to ensure that they are acting in accordance with the organisations policy, and it is the responsibility of system administrators to enforce the policy. The purpose of the access control system is simply to prevent dangerous accesses, and to provide accountability and auditability of user actions. This assumes that in general, users can be trusted to behave appropriately, that the need for users to execute questionable actions will only arise infrequently, and an administrator can recover from actions which damage the system integrity.

However, clearly such a mechanism is open to abuse if no additional constraints are placed on the actions which the user performs. If users are allowed open slather on the system, then the potential for both accidental and intentional misuse can cause serious risks.

One way to ensure that integrity is maintained in an optimistic system, is to ensure that any action that the user performs can be rolled back or compensated. This allows the system administrator to decide after-the-fact that an action was unreasonable and recover the system to a valid state. While, the cost of recovery might be high, if the instances requiring recovery are relatively infrequent, this cost will be balanced by the increased empowerment of users.

### 2.3 Summary

Balancing a security policy between the needs of users, and a need to secure the system is a difficult task. Security administrators need to consider not only the appropriate authorisations for the general running of the system, but what is appropriate in unforeseen circumstances. In addition, administrators may need to contend with an access control system that is not sufficient to support complex authorisation semantics, and need to consider how to give controlled access to certain resources which exceed a user's normal privileges.

While the Unix operating system currently provides some limited support for the later, it provides no optimistic method by which users can gain privileges in unforeseen circumstances. Thus, a mechanism is needed to provide better support for both legitimate and questionable accesses for Unix users.

## 3 Formal Model

### 3.1 Clark-Wilson Integrity Model

In their seminal paper [5], Clark and Wilson argued that unlike military systems whose main aim is to prevent disclosure, the goal of commercial security systems is to ensure that the integrity of data is preserved. They define the concept of a *well-formed transaction* as a transaction where the user is unable manipulate data arbitrarily, but only in constrained ways that preserve or ensure the integrity of the data. A security system in which transactions are well-formed ensures that only legitimate actions can be executed.

Clark and Wilson's paper presents a formal model for data integrity which consists of a number of components:

**Constrained Data Items (CDIs)** Objects that the integrity model is applied to.

**Unconstrained Data Items (UDIs)** Objects that are not covered by the integrity policy (eg. information typed by the user on the keyboard).

**Integrity Verification Procedures (IVPs)** Procedures for verifying that CDIs conform to the integrity policy.

**Transformation Procedures (TPs)** A procedure which transforms a CDI from one valid state to the other. This is the Clark-Wilson concept of a well-formed transaction.

Clark and Wilson's paper presented nine rules for the enforcement of system integrity which are listed in Figure 1.

| Rule | Description |
|---|---|
| C1 | IVPs must ensure that CDIs are valid |
| C2 | TPs on CDIs must result in a valid CDI |
| C3 | Separation of Privilege & least privilege |
| C4 | TPs must be logged |
| C5 | TPs on UDIs must result in a valid CDI |
| E1 | Only certified TPs can operate on CDIs |
| E2 | Users must only access CDIs through TPs for which they are authorised |
| E3 | Users must be authenticated |
| E4 | Only administrator can specify TP authorisations (i.e. MAC) |

Figure 1: Clark-Wilson Integrity Rules

The rules are divided into two types: certification and enforcement. Both involve ensuring compliance with the integrity policy. But certification involves the evaluation of transactions by an administrator, whereas enforcement is performed by the system.

Examination of the Clark-Wilson model, shows that the nine rules seek to enforce four separate, but related security properties:

**Integrity** An assurance that CDIs can only be modified in constrained ways to produce valid CDIs. This property is ensured by the rules: C1, C2, C5, E1 and E4.

**Access control** The ability to control access to resources. This is supported by the rules: C3, E2 and E3.

**Auditability** The ability to ascertain the changes made to CDIs and ensure that the system is in a valid state. This is ensured by the rules C1 and C4. However, in the Clark-Wilson model, log data is modelled as a CDI, which means the rules for integrity also apply.

**Accountability** The ability to uniquely associate users with their actions. This requires authentication of such users which is enforced by rule E3.

### 3.2 Partially-Formed Transactions

While well-formed transactions provide integrity assurances for legitimate actions, they do not allow for the possibility of questionable actions. As indicated in section 2.2, such actions require a relaxation of some constraints, and a mechanism which allows recovery of the system, in the event of failure.

The concept of a *partially-formed transaction*[16] can be used to describe transactions where the integrity of the data is not guaranteed, but where a compensating transaction exists to return data to a valid state. The transaction is said to be partially-formed, as the integrity of the system is only guaranteed by the compensating transaction, and not by constraining the actual action itself.

To support partially-formed transactions, a set of rules is needed to describe how such transactions can be constrained. Like those for well-formed transactions, these rules must ensure the integrity, access control, auditability and accountability of the transaction is maintained. These rules are summarised below.

**C1** IVPs must ensure that all CDIs are in a valid state at the time the IVP is run.

**C2** All TPs must be certified to provide a compensating TP which will return any modified CDI to a valid state.

**E1** The system must ensure that only TPs which have been certified against requirement C2 are allowed to run.

**E2** The system must ensure that users can only use those TPs for which they have been authorised.

**E3** The system must authenticate the identity of each user.

**E4** Each TP must write to an append-only log all the information required to reconstruct the operation, along with the corresponding compensating TP to reverse it.

**E5** Only an administrator is permitted to authorise users to access TPs (Mandatory Access Control).

Rule C1 is needed (as in the Clark-Wilson model) to determine whether or not a CDI is in a valid state in order to satisfy the auditability requirement. Rule C2 certifies that the TP corresponds to notion of a partially-formed transaction. Rule E1 exists so that only TPs which can be recovered from are allowed to be executed, and rules E2 and E3 provide the accountability and access control requirements. Rule E4, which provides the accountability requirement, is specified as an enforcement rule rather than a certification rule (contrary to the Clark-Wilson model). This is needed, as it is not possible to model the log as a CDI in the partially-formed transaction model because recoverability relies on this logging information to be kept secured.

### 3.3 Compensating actions

One issue is whether compensating actions themselves should be well-formed or partially-formed. If they are well-formed, then they will consider the data item being recovered to be a UDI which must be converted to a CDI. Having compensating actions which are well-formed mean that the system may always be rolled back to a valid state.

However, if compensating actions are only partially-formed, this provides the ability to support undo-redo semantics. It may be that the compensating actions themselves are questionable, and a system administrator may wish to undo the recovery. There are benefits to both ideas, and as such the question should be left open for the individual system designer to decide.

### 3.4 Fit-for-purpose

Whether partially or well-formed transactions should be used depends on the particular environment we are trying to secure. In the case of the accounting or financial systems on which Clark and Wilson's model was based, the strong integrity assurance obtained by using well-formed transactions are needed, as the benefits of providing an optimistic access control system are outweighed by the risks of fraud. However, in the case of highly collaborative and time-critical environments (e.g. health care), it may be more important to support the optimistic system, and use the compensating transactions to clean up the mess later.

In practice, it is likely that both types of transaction will co-exist. Security administrators may wish to enforce a general integrity policy using well-formed transactions, but provide entry points for a subset of partially-formed transactions to allow for unforeseen circumstances. Execution of partially-formed transactions can then be carefully monitored to ensure that they are only used in appropriate circumstances.

Partially-formed transactions may also be useful in the case where the integrity of a TP cannot be easily certified, but the ability to reverse its effects on the system can be. For example, an untrusted application which is allowed write access to the filesystem can be allowed to run as a partially-formed transaction providing the system is able to detect the changes made to files and provide a way to reverse them.

### 3.5 Summary

The Clark-Wilson integrity model provides a means by which a system can be constrained to ensure that only legitimate accesses can be executed. However, for the reasons given in section 2, it appears that there is a need to provide a controlled way for relaxing these restrictions to support user's abilities to perform their work. The notion of a partially-formed transaction provides a mechanism by which a system can seek to be optimistic about authorising questionable user actions.

## 4 Design

It appears on even the most cursory examination of the Unix operating system that it is incapable of enforcing partially-formed transactions, and that many standard Unix applications have not been certified to provide the logging requirement necessary for well-formed transactions. In the following sections, the design and implementation of a prototype is described which enforces these rules for the Unix operating system.

In designing a secure mechanism for enforcing well-formed and partially formed transactions, a number of issues had to be considered:

**Security** This is the first and foremost requirement for the system. The design must ensure that the rules for well-formed and partially-formed transactions are adhered to enforced, and that the mechanism does not create any other vulnerabilities for the system. Standard good programming practices, such as those described in [9] and [4] need to be adhered to, and the prototype should reflect basic, secure design principles such as those described in [18].

**Flexibility** The system must provide an expressive policy mechanism which enables administrators to configure the mechanism for a range of applications.

**Simplicity** The design and implementation should be as simple as possible to ensure that it can be easily understood and evaluated. In addition, any policy mechanism must be easy to use and understand.

**Extensibility** The design should be modular and easy to extend.

### 4.1 Reference Monitor

The design of the secure mechanism is based on the reference monitor concept described in [21]. A reference monitor is "an abstract machine that mediates all accesses to objects by subjects" [14]. Figure 2 shows how this reference monitor was implemented.

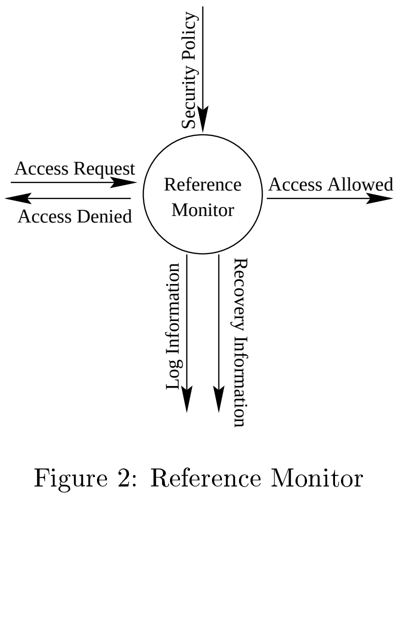


Figure 2: Reference Monitor

The reference monitor takes as input a security policy expressed in a high level language, and a particular action which is being performed. It first ensures that the user is authenticated, then evaluates the security policy to see whether the action complies. If the action is accepted, then recovery information may be stored to roll back the action at a later date. If it is rejected, the access is aborted and an error returned In either case, information about the access may be appended to a log file. The security of the reference monitor is critical as it will be solely responsible for ensuring that the rules specified in section 3 are enforced.

### 4.2 Logging and Recovery

Well-formed and partially-formed transactions require the accumulation of log information for audit and recovery purposes. This log information must be kept secured, and the implementation should provide a mechanism by which the level of logging and recovery can be specified. The implementation should also provide a mechanism for sending alerts for dangerous accesses, either by email or logging to the system console.

### 4.3 Policy Language

Designing a system for enforcing both well-formed and partially-formed transactions requires building a mechanism which not only enforces the necessary rules, but provides a rich way of expressing the policy mechanisms. As discussed in section 2, one of the problems with the Unix operating system is that its access control mechanism is too simplistic to cope with complex access control policies, and therefore some user actions may be classified as questionable by the system, which would be declared legitimate in a more flexible system.

The aim of the policy language is to be simple yet flexible enough to support rich access control paradigms such as Role-Based Access Control and Domain and Type Enforcement (see sections 5.5.2 and 5.5.3).The policy language should also support the grouping of related actions into *tasks* to provide a simple mechanism for specifying authorisations based on a number of finer grained actions.

In addition the policy language should be flexible enough to support the addition of future actions or constructs.

### 4.4 Types of actions supported

In the initial prototype, only control of filesystem actions will be supported. While this is appropriate for an initial proof-of-concept, in a robust system other actions such as networking and interprocess communication should be controlled. Actions on which authorisations can be based should include standard read, write and execute privileges, along with append-only, delete, truncate, and changing of file status information.

## 5 Implementation

### 5.1 Overview

`tudo` (Trusted User Do) is an application designed to enforce both well-formed and partially-formed transactions in a Unix operating system. It is based on `sudo`[19], originally developed at SUNY-Buffalo and since enhanced by numerous contributors. `sudo` provides a way to give users restricted root access by allowing them to execute certain programs with superuser privileges. `tudo` builds on this idea to support fine-grained access control of files and directories, and to provide the logging and recovery features necessary to support well-formed and partially-formed transactions. Despite being based on `sudo`, `tudo` has been written completely from scratch to provide these features.

`tudo` has been implemented on the Solaris operating system, although it has been designed to be portable to other Unix variants. The approach used to provide a reference monitor is similar to that taken in Janus (an environment for untrusted applications) [11]. `tudo` uses the system call tracing facilities provided by the `/proc` filesystem in the Solaris operating system to trap access to certain system calls and ensure that this access complies with the specified security policy. `tudo` forks a child to perform the requested action, and traces this process and its descendants to ensure they comply with the policy. Unlike Janus however, `tudo` allows recovery from certain actions by logging information to enable recovery in the case of failures.

tudo runs as a setuid root application. Actions are invoked, either by specifying command line arguments to execute, or giving a task or role the user wishes to perform (see below for a more detailed explanation of these constructs). When tudo is started, it first prompts the user for their password and then checks to see if the requested action is authorised. For example, a user wishing to edit the line printer configuration file might invoke:

```
$ tudo vi /etc/lpd.conf
Password:________
...
"/etc/lpd.conf" 99 lines, 2946 characters
```

Then restart the printer daemon using the `lpc restart` command (a normal setuid application).

tudo also provides the ability to group a number of related actions together in a *task*. Tasks provide controlled entry points to privileges, as the task can specify that a specific command and arguments must be run. Tasks can also include other tasks to enable new authorisations to be constructed from smaller building blocks. For example, an administrator might construct a `reboot` task to allow users to reboot the system. Authorised users could then invoke the shutdown by referring to the task rather than the command line arguments: e.g

```
$ tudo -t reboot
Password:________

Shutdown started. Thursday March  4 19:30...
```

Tasks can also be dynamically invoked, when tudo executes a command for which a particular task is defined. When a task is dynamically invoked, it is run with the privileges specified for the task, rather than those specified for the user.

Lastly, tudo provides the ability to specify *roles* to associate principals and authorisations. A role includes a list of the tasks and authorisation rules specific to that role, along with a list of users, groups, and other rules which may assume it. A user may invoke a shell as a given role using the `-r` flag to tudo. For example, a role might be created to allow its members to remove temporary files if the filesystem is filling up.

```
$ tudo -r SpaceCleaner
Password:________

# find . -name core -print | xargs rm
# find /home -name '*~' -print | xargs rm
# rm -rf /home/*/.netscape/cache/
```

## 5.2 Access Control

When a system call trap occurs, tudo consults its list of access rules to determine if the access is allowed. If it is, tudo does any necessary log and recovery processing and tells the process to continue executing. If the action is not allowed, the process is directed to abort which causes the system call to be interrupted with an EINTR errno. Because some applications attempt to retry system calls which fail with this error, tudo supports a feature from the Janus system which detects a number of failed system calls with the same parameters and then kills the process.

Because a possible race condition may exist, tudo also checks the result of the system call in some circumstances (e.g. opening a file), to ensure that the state of the object being accessed hasn't changed between the access control decision and execution of the system call. If tudo finds that the object has changed, it immediately kills the process by sending it a SIGKILL signal and sends a message to the log. tudo also checks the result of system calls when recovery and logging is turned on to determine if the action was successful or not.

## 5.3 Secure Logical Partition

Because the integrity of the configuration, log and recovery information is crucial to the security of the system, tudo maintains a *secure logical partition* where these files are kept. This is a logically separate portion of the filespace which has been set aside specifically for that purpose. Access to the secure partition is automatically denied by implicit rules unless the user has specific authorisation as a tudo administrator.

## 5.4 Recovery

tudo provides a very simple recovery feature which allows certain modifications to the file system to be rolled back. tudo allows the administrator to specify whether or not certain actions can be rolled back, and can specify what the default policy is for particular tasks, roles or for all users. tudo only allows recovery from an entire session (i.e. the effect on the system from when tudo is invoked to when the last child it is tracing exits). This simplifies the process of recovery, as tudo does not need to worry about serialising the order of events which occur within the session[1]. tudo compares the states of accessed files and directories at the start and end of a unit of work and logs enough information to reverse any changes made. tudo provides utilities to examine the event log, and roll back individual actions. Rollback in tudo is implemented in the following manner:

- When an `open()` system call is invoked, tudo first checks to see if the access is allowed (depending on the flags). If the argument to `open()` is a regular file or a directory, and is being modified, tudo then makes a backup copy of the file on the secure partition. When the session completes, tudo creates a context diff of the modified file using the `diff` command. This context diff can then later be used as input to the `patch` command to restore the file. In the case that the modified file is a binary file, the original backup is kept. In either case, if tudo detects that the file has not changed then it logs this fact and removes the backup copy.
- A `creat()` system call, or an `open()` with `O_CREAT` which creates a new file is rolled back by deleting the created file.
- A `rename()` call is rolled back by changing the old filename back to the new filename.
- When a `chmod()`, `chgrp()` or `chown()` is performed on a file or directory, tudo first calls `stat()` on the file and saves the old file permission/ownership information. Rollback is performed by restoring the permissions to their original state.
- When a file is deleted using the `unlink()` call, tudo makes a copy of the file on the secure partition. However, often administrative task involve removal of large amounts of unnecessary files (e.g. rm /tmp/*), so setting a recovery option for removals can result in a large amount of diskspace being consumed. For this reason, recovery of unlinked files should be avoided. Where possible, administrators should rely on backups to provide recovery for removed files. It should be noted that tudo will not backup the file if there is a hard link to the file somewhere else on the filesystem, but instead logs the inode of the link.

[1] However the issue of serialising the order of multiple tudo sessions is not considered (see section 6.4.1

If a tudo session exits unexpectedly without cleaning up the recovery information this will be detected the next time tudo is invoked, and steps will be taken to clean up the session.

A recovery action runs as a normal tudo action, so this also provides redo semantics. However, it should be noted that users will not always have the necessary privileges to rollback an action, so this may have to be performed by an administrator.

## 5.5 TUPL

### 5.5.1 Overview

tudo uses a simple language called TUPL (Trusted User Policy Language) to express access control policies. TUPL supports several constructs which allow the easy expression of policies to support well-formed and partially-formed transactions. TUPL allows the administrator to define access rules which associate a collection of users, groups or roles with an action. These rules take the form:

```
allow <action> <args> [by <principal>]
    [ "[" options "]" ]
deny <action> <args> [by <principal>]
    [ "[" options "]" ]
```

The `principal` may specify a user, a group or a role, or a list of these. An `<action>` may be a list of tasks, predefined actions or the special action `any`, which means that the access applies to any action. Figure 5.5.1 lists the predefined actions which are valid. The design of the policy module is such that more actions can be easily added.

| Target | Actions |
|---|---|
| files | `read, write, modify, append, create, trunc, rm, exec, rename, link, chmod, chown, chgrp` |
| dirs | `ls, chdir, rmdir, chgrp, chown, chmod` |

In addition, TUPL allows a number of constructs to be specified which can be combined with the access rules to implement more complex policies. TUPL supports the following constructs:

`type` Specifies a class of objects for which particular rules can be enforced. Currently only the `file` and `dir` classes are supported, although the policy module is designed so that new class types can be added. e.g. all files in the CVS directory owned by the engineering or research group can be specified using:

```
type project_code {
    class: file;
    path: "/usr/local/cvs/*";
    owner: (engineering, research);
}
```

`task` Identifies a particular task type which might be allowed, and the actions which should be allowed to be performed when executing that task. The task has an optional field `command`. When `tudo` is executed with the `-t` option, and a task is specified, the command and arguments in this field will be executed, or in the case that none is present, `tudo` will fork a shell. If a task is not specified in this way, then it may be dynamically invoked when `tudo` encounters a request to execute a program which matches the `command` field. e.g. a task which allows users to edit the line printer configuration files can be specified using the following construct:

```
task edit_lpd_conf {
  command: "vi /etc/lpd.conf";
  description: "Edit lpd.conf";
  rules: {
    //Allow user to run vi
    allow exec "/bin/vi";

    //Allow vi to open shared libs
    //Assumes lib_path_type already
    //defined
    allow read lib_path_type [log=0];

    //Allow read/write on tmp files
    allow all "/var/tmp/*" [log=0,
       recover=no];

    //Allow read/write on lpd.conf
    allow (read, write)
        "/etc/lpd.conf"
        [log=3, recover=yes];

    //Deny everything else
    deny all;
  };
}
```

Authorisations specified in subtasks override authorisations in the parent task for the period of time which that sub-task is executing.

`tudo` tasks are similar to the process-specific file protection provided in the TRON system[3]. In TRON, system call wrappers are used to provide a protection domain in which normal processes can run with restricted privileges. Like TRON, `tudo` allows rights to be specified on a per process basis (as identified by the program and arguments used to create that process), but also provides the capability to roll back actions.

`role` A list of users and groups along with their access privileges. e.g. The role "SpaceClearer" which comprises the engineering and admin group and are allowed to delete "junk" files to clean up space can be specified as:

```
type junk_files {
  class: file;
  path: ("/home/*/core",
         "/home/*/*~",
         "/home/*/.netscape/cache",
         "/home/*/.netscape/cache/*");
};

role SpaceClearer {
  members: (engineering, admin);
  rules: {
    allow search "/home/*";
    allow rm junk_files;
    allow exec ("/bin/find",
                "/bin/xargs",
                "/bin/rm");

    //Everything else is disallowed
```

```
    deny all;
    }
  };
```

### 5.5.2 Role-Based Access control

RBAC is an access control paradigm in which access decisions are based on the functions a user is allowed to perform within an organisation, rather than "data ownership" [7]. RBAC is supported in TUPL by allowing the administrator to define roles using the `role` construct. A role can include users, groups or other roles as members. Each role specifies the access control rules associated with it. In addition, other access control rules can be defined to apply to roles. TUPL does not however, support conditional membership rules, and roles are not dynamically invoked. A user must specify which role they are adopting when invoking a `tudo` session.

### 5.5.3 Domain and Type Enforcement

Domain and Type Enforcement (DTE) allows objects and subjects in the system to be typed, and authorisations to depend on these types [1]. Subjects perform actions in *domains*, and the DTE system provides rules for how transitions to new domains can occur. DTE is supported in TUPL using the `type`, and `task` constructs. In TUPL, the concept of a `task` roughly equates to a DTE domain, and the `type` constructs equate to a DTE type. By allowing tasks and sub-tasks to be dynamically invoked, TUPL supports a limited way of constraining transitions between domains. However, this is not as flexible as that supported in other DTE mechanisms (e.g. [1]).

# 6 Evaluation

In order to determine the usefulness of the `tudo` utility, it must be evaluated to determine how well it meets the design criteria in section 4. In particular, it must enforce well-formed and partially-formed transactions in accordance with the models presented.

## 6.1 Enforcement of well-formed transactions

Providing well-formed transactions requires the enforcement of the four rules described by Clark and Wilson.

- E1 – Only certified TPs can operate on CDIs

  In order to perform an action it must be specifically authorised in the policy file. Because only certified actions are assumed to be authorised, this meets the requirement for rule E1.

- E2 – Users must be authorised

  This rule is also enforced by the policy mechanism.

- E3 – Users must be authenticated

  This rule is enforced by requiring the user to enter their password in order to perform an action.

- E4 – Only administrator can specify TP authorisations (i.e. MAC)

  `tudo` provides a specially administrator privilege which is required to edit the policy file to ensure this rule is enforced.

## 6.2 Enforcement of partially-formed transactions

Enforcement of partially-formed transactions requires that the system observe the five enforcement rules described in [16].

- E1 – The system must ensure that only TPs which have been recoverable can be run

  This rule is enforced for all actions where the recover=yes option is set. If recover=yes is set for an option which is not recoverable, that action will fail (this can happen for example when using the rule `allow all [recover=yes]`.

- E2 – Users must be authorised

  This is enforced by the policy mechanism.

- E3 – Users must be authenticated

  This is enforced by requiring a user to enter their password in order to perform an action.

- E4 – Transactions must be logged

  This is enforced by the logging mechanism in `tudo`.

- E4 – Only administrator can specify TP authorisations (i.e. MAC)

  `tudo` provides a specially administrator privilege which is required to edit the policy file to ensure this rule is enforced.

## 6.3 Security

One of the most important lessons learned in implementing `tudo` is that the *principal of least privilege* is sagely advice. Implementing an optimistic system which relaxes this requirement raises many challenges. These issues are discussed below.

### 6.3.1 Programming practices

`tudo` has been implemented using good programming practices to ensure that running it as a setuid application does not open up any security vulnerabilities. In particular `tudo` ensures that all system call returns are checked, and that only "safe" version of library functions for reading and manipulating user input are used. In addition, `tudo` sanitises environment variables by setting them to safe values.

### 6.3.2 Self protection measures

It should be noted that the enforcement mechanism that `tudo` uses is rather fragile. If a user can find a way to kill the tracing process without killing the programs it is tracing, then they will be able to perform actions without being subject to the access control mechanism. To help prevent this, `tudo` sets the *kill-on-last-close* flag for each process it traces, so that if `tudo` exits for some reason, all traced processes will be automatically killed with a SIGKILL signal. In addition, `tudo` disassociates itself from its parent process group to ensure that killing parent processes does not kill `tudo`. Finally, `tudo` contains an implicit rule which prevents any traced application from sending a SIGKILL signal to any `tudo` process, or from opening any file in a `tudo` processes `/proc` filespace for write access.

### 6.3.3 Race conditions

As discussed in section 5.2, a potential race condition exists between checking the authorisation and allowing the system call to proceed. `tudo` protects against this by checking the status of some objects after the system call has completed and killing the process if they do not match expectations.

### 6.3.4 Handling a full secure logical partition

`tudo` has a hardcoded high watermark on the capacity of the logical partition, and will abort a session and roll back the intermediate states once this high watermark is exceeded. This stops a user who might have write permission to the physical volume on which the secure logical partition is stored from filling up the disk in order to prevent `tudo` from logging information. Once the high watermark is exceeded, `tudo` requires that log state be cleaned up to below a low watermark before any more activity is allowed (with one exception - to allow the administrator to specify `tudo` tasks which cycle the logs etc, `tudo` will allow those actions/task with the option `cleanup=yes` set to operate while the high watermark has exceeded).

### 6.3.5 Handling symbolic and hard links

Handling of filesystem links is somewhat complex in `tudo`. Because file authorisations are done on the basis of filename patterns, it could be possible for a user to create a symbolic link to a restricted file in an unrestricted filespace. For this reason, `tudo` always applies access restrictions on the real filename for symbolic links.

A more complex situation occurs for hard links where two different files share the same inode on the same filesystem. It is difficult for `tudo` to determine the names of all files linked to a given inode without performing a scan of the raw disk device. For this reason, `tudo` allows accesses on files with multiple hard links to occur, but logs a warning message to indicate that the access is potentially suspicious.

#### 6.3.6 Constraining non-syscall access

The Unix operating system provides other ways to manipulate the system other than system calls. Pseudo files such as `/dev/kmem`, `/dev/mem`, `/proc` and the the raw disk device files can provide ways for users to bypass the security mechanisms provided by `tudo`. Again, because such accesses can be legitimate, the administrator must explicitly deny users access to such files through a rule such as:

```
deny all ("/dev/*", "/proc/*");
```

### 6.4 Limitations

There are some limitations with the `tudo` prototype. Using the system tracing facility of the `/proc` filesystem means that a file descriptor must be open for every process being traced. If a large number of processes are being executed (for example in a shell script), `tudo` may be unable to complete the task.

In addition, determining the privileges necessary to complete a particular task can be time-consuming and difficult. An administrator can simplify the process by creating a task with a command entry point and authorising all actions for that command, but this creates problems if the command is not entirely trusted. Often an administrator will need to use a system call tracing facility such as `strace` or `truss` to determine what accesses an application needs.

A further limitation is that `tudo` does not provide the ability for administrators to specify how actions should be rolled back. Currently, recovery is implemented by tracking changes to modified files and reverse applying the changes. However, there may be some higher level semantics which require such actions to be compensated in a different way.

#### 6.4.1 Lack of transactional semantics for `tudo` actions

Perhaps the biggest limitation of `tudo` is that it does not provide full transactional semantics (i.e. ACID properties) for actions, and as such it is possible for programs to either see or modify intermediate states. This arises for two reasons:

1. `tudo` itself does not perform any isolation or concurrency control of `tudo` sessions, and as such two `tudo` sessions running at the same time may create conflicts.
2. Users may perform other operations without using `tudo`, but which either read or write objects accessed in `tudo` sessions. As these actions are neither logged nor recoverable, a user can use their `tudo` privileges to engage in seemingly legitimate behaviour, but can run a simultaneous session which exploits insecure intermediate states.

This problem is indicative of the lack of transactional semantics in Unix in general. Because the security of both well-formed and partially-formed transactions are based on the assumption of atomic and isolated actions, this short-coming is problematic for `tudo`, and can lead to situations in which privileged access can bypass the reference monitor.

Of the two reasons given above, the second case is perhaps the most difficult for an optimistic system. Recall that for partially-formed transactions, consistency is only guaranteed by the existence of a compensating mechanism which can be used to recover the system to a valid state. If we compare this situation to a traditional database system, we see that the commit phase is analogous to a system administrator reading a `tudo` log, and judging whether an action should be accepted or rolled back. However, because `tudo` exposes the intermediate state, it may be the case that either rolling back will not undo all the damage that has been done, or that the administrator will be unaware that a conflict exists.

A simple example which illustrates this problem is when a user creates a setuid executable, or downgrades the file permissions on an important file (`/etc/passwd` for example) using `tudo`. While `tudo` can roll back these actions, the system administrator can never be sure what the user did while these backdoor holes were in place. The user might legitimately argue that the permission change was an accident and that there was no malicious intent.

The solutions to the first problem are relatively straight forward applications of traditional concurrency control. `tudo` could implement its own optimistic timestamping mechanism and use its existing rollback mechanism to cope with conflicts. However, while the solutions to the second problem are simple in theory, they are pragmatically less desirable. One approach would be to require that all users run `tudo` as their default shell so that all actions are logged and recoverable. This doesn't close all the holes (e.g. setuid programs which are able to leverage root privilege to work around `tudo`'s controls as described above are still a problem, and a user might still be able to rewrite their shell in the password file), nevertheless, it does improve the situation. However, users may be unwilling to pay the inevitable performance penalty which comes with `tudo` for resource/CPU intensive applications. Another approach which is maybe more palatable from this aspect, but at the cost of reducing the flexibility of the mechanism, is to write rules into `tudo` to ensure that no object can have its privileges downgraded and no principal can have their privileges upgraded *except* during a given `tudo` session. This would mean any changes to a file's permissions for example, would have to be undone before a `tudo` session exited, otherwise the entire session would be aborted and the actions would be rolled back.

### 6.5 Other uses

This paper has described the use of `tudo` to enforce well-formed and partially-formed transactions. However, `tudo` can also be used for other purposes. `tudo` can also provide a sandbox for untrusted applications such as that provided by the Janus environment[11]. In addition, `tudo` could also be used to run setuid shell scripts, similar to `super`[20].

## 7 Related Work

In [5], Clark and Wilson compare their integrity model to others based on the Bell and LaPadula secrecy model [2]. Because partially-formed transactions are closely related to well-formed transactions, much of this discussion also applies to them.

Foley[8] examines various integrity models (including Clark-Wilson) and argues that these models are limited in the sense that they only consider integrity in an operational/implementation sense. Foley presents a formal model in which integrity is considered as just one attribute of a *dependable* system. He shows that dependability can be seen as a form of refinement, in which the system can implement top-level requirements in the presence of failures, and demonstrates how this model can be used to describe concepts such as separation of duties, and assured pipelines as implementation techniques for achieving integrity. Because partially-formed transactions are simply another such implementation technique, they could be easily modelled using Foley's scheme.

As discussed, Goldberg et al [11] describe an environment similar to `tudo` which is used to provide a *sandbox* for untrusted helper applications. While their design approach is similar to that taken by `tudo`, Janus differs in the fact that it does not provide the logging and recovery features which are central to `tudo`'s architecture.

Numerous access control systems have been implemented to augment Unix's simple system, some of which are discussed in [11]. Notable works include: [3, 1, 6]. These systems are primarily aimed at providing richer access control semantics for Unix, and do not provide the recovery features necessary to implement partially-formed transactions. These systems have also been implemented using kernel modifications, or as system call wrappers which would seem to be a less flexible (although possibly more performant) mechanism than is used for `tudo`.

Recovery has been well studied in the area of concurrency control and transaction processing. Ramamritham and Panos [17] give an excellent overview of the state of the art in this area. Partially-formed transactions are similar to *sagas*[10], as they are long running, and consist of a number of independent component transactions. Like partially-formed transactions, compensating actions are used in sagas to maintain consistency. Partially-formed transaction also have something in common with *transactional workflows*, where a desire to relax the ACID properties of a transaction mean that system failures must be dealt with using compensating transactions [13]. It would be useful to investigate the techniques used in these more exotic transactional systems to see how they might improve some of the limitations with the `tudo` prototype.

## 8 Future Work

This paper has focussed on the issues of how to provide both pessimistic and optimistic access control using well-formed and partially-formed transactions in a Unix environment. However it would be useful to extend this work by looking at how different authorisation models such as $n$ of $m$ threshold schemes might help to minimise the risks of misuse by requiring more than one person to request a questionable action before it is authorised.

Implementing rollback for network and interprocess communications is another interesting problem, as compensating transactions in this case are not clear cut. Some investigation of how to recover from such actions would help to extend the concept of partially-formed transactions to these domains.

Finally, performance issues in the `tudo` prototype should be addressed. These are particularly problematic in the case of modifying very large binary files, as `tudo` currently needs to take a full copy of these files each time they are opened for write. It would also be useful to see how `tudo` could work with a log-type filesystem where the ability to roll back changes is supported by the filesystem, rather than as an add on using `tudo`.

## 9 Summary

This paper has motivated the need for a mechanism to allow users to not only perform legitimate actions, but to allow for cases in which certain questionable actions should be authorised. It describes an optimistic access control mechanism which seeks to maintain the integrity of transactions by providing a mechanism which allows actions to be rolled back to a valid state. The concept of partially-formed transactions has been formally described, and the paper shows how such transactions can be used to enforce the security properties of integrity, access control, audibility and accountability for an optimistic system.

A proof-of-concept prototype has been designed and implemented for the Solaris operating systems. It demonstrates how a reference monitor can be constructed using the system call tracing facilities of the `/proc` filesystem which enforces both well-formed and partially-formed transactions. Security features and limitations of the prototype are discussed, and future directions are discussed for this research.

## 10 Acknowledgements

The work reported in this paper has been funded in part by the Co-operative Research Centre Program through the Department of the Prime Minister and Cabinet of Australia.

## 11 Availability

The `tudo` prototype is available from

http://security.dstc.edu.au/projects/tudo/

## References

[1] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghihat. Practical domain and type enforcement for UNIX. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 66–77, 1995.

[2] D. E. Bell and L.J. LaPadula. Secure computer systems. Technical Report ESD-TR-73-278 (Vol I-III) (also Mitre TR-2547), Mitre Corporation, Bedford, MA, April 1974.

[3] Andrew Berman, Virgil Bourassa, and Erik Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the 1995 USENIX Winter Technical Conference*, pages 165–175, 1995.

[4] M. Bishop. Unix security: Security in programming. *SANS'96*. Washington DC, May 1996. URL: http://seclab.cs.ucdavis.edu/-bishop/secprog.html.

[5] David D. Clark and David R. Wilson. A comparison of commercial and military security policies. In *1987 IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, CA, April 1987.

[6] G. Fernandez and L. Allen. Extending the Unix protection model with access control lists. In *Proceedings of the Summer 1998 USENIX Conference*, pages 119–132, 1988.

[7] David Ferraiolo and Richard Kuhn. Role-based access control. In *Proceedings of the 15th National Computer Security Conference*, 1992.

[8] Simon N. Foley. Evaluating system integrity. In *New Security Paradigms Workshop*. ACM press, 1998.

[9] Peter Galvin. The Unix secure programming faq - tips on security design principles, programming methods and testing. *Sunworld*, August 1998. URL: http://www.sunworld.com/swol-08-1998/swol-08-security.html.

[10] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 249–259, New York, NY, 1987. ACM press.

[11] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications – confining the wily hacker. In *Proceedings of the Sixth USENIX Security Symposium*, San Jose, California, July 1996.

[12] John D. Howard. *An Analysis Of Security Incidents On The Internet:1989 - 1995*. PhD thesis, Carnegie Mellon University, April 1997. URL: http://www.cert.org/research/JHThesis/Start-.html.

[13] Dean Kuo, Michael Lawley, Chengfei Liu, and Maria Orlowska. A model for transactional workflows. *Australian Computer Science Communications*, 18(2):139–146, 1996.

[14] Dennis Longley, Michael Shain, and William Caelli. *Information Security – Dictionary of concepts, standards and terms.* Macmillan, 1992.

[15] Peter G. Neumann. *Computer related risks.* Addison-Wesley, 1995.

[16] Dean Povey. Optimistic security: A new access control paradigm. In *Proceedings of the 1999 New Security Paradigms Workshop*, September 1999. to appear.

[17] Krithi Ramamritham and Panos K. Chrysanthis. *Executive Briefing: Advances in Concurrency Control and Transaction Processing.* IEEE Computer Society Press, Los Alamitos, CA, 1997.

[18] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[19] sudo. URL: ftp://ftp.courtesan.com/pub/sudo.

[20] super. URL: ftp://ftp.ucolick.org/pub/users/will/.

[21] *Department of Defense Trusted Computer System Evaluation Criteria.* Department of Defense Computer Security Center, Fort Meade, MD, August 1983.