TEAM 2: PROBLEMS IN COMPUTER SECURITY

AUERBACH, KERBEL, MEGRAW, OSBURN, SHETTY[1]
with mentor John Hoffman[2]

August 30, 1998

[1]IMA, University of Minnesota
[2]Secure Computing Corporation

0.1 Abstract

We consider an intransitive non-interference security policy for a system that can be represented as a deterministic state machine. After defining the system and communication policy, we build an equivalence relation by analyzing the system's security using a constructive algoritm. We define a convert channel and make changes to eliminate them. We find that the policy and equivalence relation then satisfy the conditions of the unwinding theorem, thus ensuring security. Finally we formally verify the state's security in some cases using a computer package, called PVS.

Chapter 1

BASIC CONCEPTS

1.1 Definition of Security

We will first define what are the possible components of a computer system in general and then give the definition of security.

Modeling a computer system: A system M can consist of:

- a set S of $STATES$ where $s_0 \in S$ is an initial state,
- a set D of domains,
- a set A of actions,
- a set O of outputs,
- function *step*: S x $A \rightarrow S$, where $step(s, a)$ denotes the next state of the system after applying action a,
- function *output* :S x $A \rightarrow O$, where $output(s, a)$ denotes the result returned by the action a,

An example of action could be a "write" command to the file.
We also define function *run*:

- function *run*: S x $A^* \rightarrow S$ which is a natural extension of function step to sequence of actions

$$run(s, \phi) = s$$

where ϕ is empty sequence of actions

$$run(s, \alpha \circ \alpha) = step(run(s, \alpha), \alpha)$$

Communication and Security. We say that two domains u, v *communicate* ($u \hookrightarrow v$ interfere) if there is an information flow channel between them. Using

the definition of *communication* we say that a *security policy* of the system is a set of rules defining what domains can communicate.

The system is *secure* if the given *security policy* of the system completely defines all possible communication channels. If the channel of information flow was discovered that is not specified in the *security policy* of the system then we call it a *covert channel*.

1.2 Transitivity vs. Intransitivity.

An example of channel - control security policy is shown on the picture below, where machines represent security domains and edges represent the information flow that is allowed in the system. Many channel-control policies are intransitive: the Mail Server can not accept e-mail without filtering mail for junk e-mail or binary files, depending on the company policy. So information cannot flow directly from Firewall to Mail Server like it would be in transitive system.

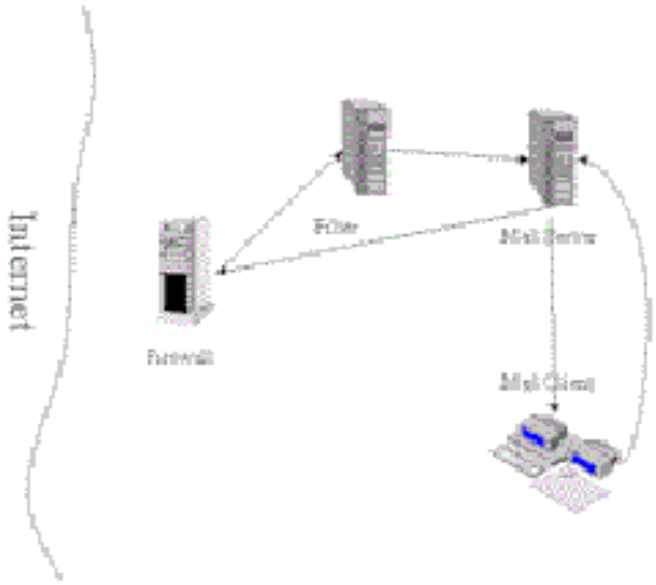

Figure 1.1: Example of intransitive computer system.

One of the differences between transitive and intransitive systems is in transitive system it is not possible to create a cycle of information flow, but it is possible in intransitive system. In the figure, there is a cycle of information flow: Firewall- Filter - Mail Server - Firewall. It is clear that Firewall interferes with Mail Server but we need to find a way of saying it must occur only through the Filter. And this is successfully can be defined through the concept of noninterference policies.

1.3 Unwinding Theorem for Intransitives polices.

An important component of noninterference formulation of the security is "unwinding" theorem that shows: it is sufficient toanalyze behavior of individual actions rather than sequences of actions to prove the system to be secure.

Definition:

- A system M is $view-partition$ if, for each domain $u \in D$, there is an equivalence relation $\overset{u}{\sim}$ on S.
- This equivalence relation is said to be *output consistent* if

$$s \overset{dom(a)}{\sim} t \Rightarrow output(s,a) = output(t,a).$$

  Output consistency is required that two states s and t appear to be identical to domain u in terms of indistinguishable outputs they produce in response of actions from u.

- Let $\hookrightarrow$ be a policy, then we say a system M is *locally respects* if it is view-partitioned and if:

$$dom(a) \not\hookrightarrow u \Rightarrow s \overset{u}{\sim} step(s,a).$$

- Let $\hookrightarrow$ be a policy, then we say a system M is *weakly step consistent* if :

$$s \overset{u}{\sim} t \vee s \overset{dom(a)}{\sim} t \Rightarrow step(s,a) \overset{u}{\sim} step(t,a).$$

Unwinding Theorem:
Let $\hookrightarrow$ be a policy and M a view-partitioned system that is:

- *output consistent*,
- *weakly step consistent*,
- *locally respects*

then M is *secure* for $\hookrightarrow$.

The proof of the theorem is shown in the paper by Rushby.

Chapter 2

The Algorithm

2.1 METHOD BACKGROUND

The method used is similar to the approach used in the case of a transitive system described in [Oakland '90]. This is one of the optimal ways of ensuring that the conditions $[C1 - C4]$ for the theorem are met in our system. Usually some assumptions must be made concerning the reachable states of the system regardless of the initial state of the system. We will use A to denote the set of propositions that we assume to be true for each reachable state. In order that these assumptions be state invariants for the request we need to show the following:

$C1$: If A is satisfied in *state_st* , then A is satisfied in any state by applying a request.

We then consider the non-interference policy on the set of states satisfying A.In the proof we consider only one request at a time since extension to arbitrary sequences of requests follows from the Unwinding Theorem for intransitive policies. The starting point of the induction can be justified by proving that each request satisfies the following :

$C2$: (locally respects)
$C3$: (weakly step consistent)
$C4$: (is output consistent)

By the Unwinding theorem for Intransitive Policies we see that conditions $C1 - C4$ are sufficient conditions for the system to be secure.

2.2 ALGORITHM

Following is the constructive algorithm used in [Oakland] which we use to show that our system is secure:

1. We begin by hoping that we need not make any assumptions about the state i.e A_0 is the empty set of propositions.

2. Let $\overset{d}{\sim}$ be defined so that it places the minimal constraint on state st_1 and st_2 so that $C4$,the condition for output consistency still holds.

So now the initial guess for A and equivalence have been made. Since any assumption to A must be satisfied in the initial state and must satisfy C1 for each request, only conditions C2 and C3 are checked in each iteration.

3. $i = 0$

4. Repeat the following request for each operation:

(a) Try to prove $C2$ for the operation using A_i and $\overset{d}{\sim}_i$ If the proof fails then a channel has been found in the system that allows a subject to access an object where the access is not in the set of allowed accesses for the domain of the subject ,and the type of the object. A Minimum set of assumptions B is found such that

- $C2$ holds for the operation using A_{i+1} and $\overset{d}{\sim}_{i+1}$
- $C1$ holds for all of the operations with A_{i+1} in place of A_i
- The assumptions in B hold in the initial state

where $A_{i+1} = A_i \bigcup B$ and $\overset{d}{\sim}_{i+1}=\overset{d}{\sim}_i$

If such a set B can be found then replace the guesses for $\overset{d}{\sim}_i$ and A_i with $\overset{d}{\sim}_{i+1}$ and A_{i+1}

$i = i + 1$ and go back to step 4.

(b) Try to prove $C3$ for the operation using A_i and $\overset{d}{\sim}_i$.If the proof fails ,then more information about $state_1$ and $state_2$ is needed. A minimal set of additional requirements are added to $\overset{d}{\sim}_i$ to define $\overset{d}{\sim}_{i+1}$. $A_{i+1} = A_i$ $i = i + 1$ and go back to step 4.

5. If this point is reached then $C1 - C4$ hold and the system has been shown to be secure.The reason for that is the following :

We add assumptions to A only if $C1$ is valid.$C2$ and $C3$ have been verified for each operation For if not then assumptions are added to the system or we have more conditions on $\overset{d}{\sim}$ such that $C2$ and $C3$ hold.Also the method of construction ensures that if two states are equivalent with respect to $\overset{d}{\sim}_i$ then they are equivalent with respect to $\overset{d}{\sim}$.This ensures that the outputs are the same in both states.Also since the view is becoming more restrictive the previous work is not invalidated when the definition of the equivalence relation is changed.Therefore from the Unwinding Theorem for Intransitive policies our system is secure.

We want to show that the algorithm applies to intransitive systems by testing it on a simple system.

Chapter 3

The System and the Proof

3.1 System Specification

A simple system is analyzed based on the system in . The analysis of this system shows that the algorithm can prove that there exists a secure intransitive system. The following describes the system before analysis:

- The system has a collection of processes (denoted by *processes*), a collection of files (denoted by *files*), a collection of domains (denoted by *domains*), and a collection of types (denoted by *types*).
- For each process, the function $proc\_dom : processes \rightarrow domains$ associates the process to a domain.
- For each file, the function $file\_type : files \rightarrow types$ associates the file to a type.
- The system state consists of the following collection of functions:
  - $file\_lock : files \rightarrow processes \cup \{\phi\}$, which indicates which process, if any, has each file locked for writing,
  - $in\_use\_set : files \rightarrow \mathbf{P}(processes)$, which indicates the set of processes that have each file open for reading, and
  - $data : files \rightarrow data\_values$, which indicates the data in the file.

  These functions are meant to indicate the current condition of the files.
- There are eight operations: WRITE, READ, LOCK, UNLOCK, OPEN, CLOSE, TEST_LOCK, TEST_OPEN.

  These operations are defined as follows:

  Let p represent a process with $proc\_dom(p) = u$, f represent a file, d represent a data value, and st and new_st represent a states.

  $READ(p, f)(st) : state \rightarrow state$

$READ\_OUT(p, f)(st) : state \rightarrow state$

$$READ\_OUT(p, f)(st) = \begin{cases} null & \text{if } p \notin in\_use\_set(st)(f) \\ null & \text{if } R \notin acc\_pol(u, file\_type(f)) \\ data\_val(st)(f) & \text{otherwise} \end{cases}$$

$WRITE(p, f, d)(st) : state \rightarrow state$

$$WRITE(p, f, d)(st) = \begin{cases} st & \text{if } W \notin acc\_pol(u, file\_type(f)) \\ st & \text{if } file\_lock(st)(f) \neq p \\ new\_st & \text{otherwise} \end{cases}$$

where new_st is identical to st except that $data\_val(new\_st)(f) = d$

$LOCK(p, f)(st) : state \rightarrow state$

$$LOCK(p, f)(st) = \begin{cases} st & \text{if } W \notin acc\_pol(u, file\_type(f)) \\ st & \text{if } file\_lock(st)(f) \neq \phi \\ st & \text{if } in\_use\_set(st)(f) \neq \phi \\ new\_st & \text{otherwise} \end{cases}$$

where new_st is identical to st except that $file\_lock(new\_st)(f) = p$

$UNLOCK(p, f)(st) : state \rightarrow state$

$$UNLOCK(p, f)(st) = \begin{cases} st & \text{if } W \notin acc\_pol(u, file\_type(f)) \\ st & \text{if } file\_lock(st)(f) \neq p \\ new\_st & \text{otherwise} \end{cases}$$

where new_st is identical to st except that $file\_lock(new\_st)(f) \neq \phi$

$OPEN(p, f)(st) : state \rightarrow state$

$$OPEN(p, f)(st) = \begin{cases} st & \text{if } R \notin acc\_pol(u, file\_type(f)) \\ st & \text{if } file\_lock(st)(f) \neq \phi \\ new\_st & \text{otherwise} \end{cases}$$

where new_st is identical to st except that $in\_use\_set(new\_st)(f) = in\_use\_set(st)(f) \bigcup \{p\}$

$CLOSE(p, f)(st) : state \rightarrow state$

$$CLOSE(p, f)(st) = \begin{cases} st & \text{if } R \notin acc\_pol(u, file\_type(f)) \\ st & \text{if } p \notin in\_use\_set(st)(f) \\ new\_st & \text{otherwise} \end{cases}$$

where new_st is identical to st except that $in\_use\_set(new\_st)(f) = in\_use\_set(st)(f) \setminus \{p\}$

$TEST\_LOCK(p, f)(st) : state \rightarrow state$

$TEST\_LOCK\_OUT(p, f)(st) : state \rightarrow state$

$$TEST\_LOCK\_OUT(p, f)(st) = \begin{cases} null & \text{if } W \notin acc\_pol(u, file\_type(f)) \\ T & \text{if } W \in acc\_pol(u, file\_type(f)) \text{ AND} \\ & R \in acc\_pol(u, file\_type(f)) \text{ AND} \\ & file\_lock(st)(f) = p \\ F & \text{if } W \in acc\_pol(u, file\_type(f)) \text{ AND} \\ & R \in acc\_pol(u, file\_type(f)) \text{ AND} \\ & file\_lock(st)(f) \neq p \\ new\_st & \text{otherwise} \end{cases}$$

$TEST\_OPEN(p, f)(st) : state \rightarrow state$

$TEST\_OPEN\_OUT(p, f)(st) : state \rightarrow state$

$$TEST\_OPEN\_OUT(p, f)(st) = \begin{cases} null & \text{if } R \notin acc\_pol(u, file\_type(f)) \\ T & \text{if } R \in acc\_pol(u, file\_type(f)) \text{ AND} \\ & in\_use\_set(st)(f) \neq \phi \\ F & \text{if } R \in acc\_pol(u, file\_type(f)) \text{ AND} \\ & in\_use\_set(st)(f) = \phi \\ new\_st & \text{otherwise} \end{cases}$$

- There is a communication policy for the system. It is defined by the function $acc\_pol : (domains, types) \rightarrow \{W, R, WR, \phi\}$. A W indicates that processes of that domain may write to, but not read from, files of that type. A R indicates that processes of that domain may read from, but not write to files of that type. WR indicates both read and write access, and ϕ indicates neither read nor write access.

- The security policy, symbolized by $\hookrightarrow$, indicates when two domains are permitted to "communicate". Domain d "communicates with" domain u, symbolized by $d \hookrightarrow u$, when $W \in acc\_pol(d, t)$ and $R \in acc\_pol(u, t)$ for some file type t.

Through analysis of the system, we discovered that TEST_OPEN causes a covert channel. This covert channel is explicitly explained below. It was also decided that TEST_OPEN is an unnecessary operation in the system, because by simply attempting to READ the file, a process can ensure that it has the file open. So in the final system, TEST_OPEN is removed from the set of operations on the system. There is also a covert channel found in TEST_LOCK. So the final TEST_LOCK operation is different. The final version is listed below.

$TEST\_LOCK\_OUT(p, f)(st) : state \rightarrow state$

$$TEST\_LOCK\_OUT(p, f)(st) = \begin{cases} null & \text{if } W \notin acc\_pol(u, file\_type(f)) \\ null & \text{if } R \notin acc\_pol(u, file\_type(f)) \\ T & \text{if } W \in acc\_pol(u, file\_type(f) \text{ AND} \\ & R \in acc\_pol(u, file\_type(f)) \text{ AND} \\ & file\_lock(st)(f) = p \\ F & \text{if } W \in acc\_pol(u, file\_type(f) \text{ AND} \\ & R \in acc\_pol(u, file\_type(f)) \text{ AND} \\ & file\_lock(st)(f) \neq p \\ new\_st & \text{otherwise} \end{cases}$$

These are the only changes to the system necessary to have security.

3.2 The Equivalence Relation

The Unwinding Theorem for Intransitive Policies calls for a view-partitioned system. So it is necessary to formulate an equivalence relation on states for each domain d. The first step of the algorithm is to define the equivalence relation to satisfy the *output consistent* requirement of the Unwinding Theorem. Notice that the only operations that produce output are READ, TEST_LOCK, and TEST_OPEN. So the initial choice for the equivalence relation conditions are only the statements V1, V2, and R3, listed below. However, analysis of the system leads to removing R3, changing V2, and adding V3 and V4, which are also listed below. So the final (with the initial indicated) equivalence relation for domain d on states s and t is $s \overset{d}{\sim} t$ if:

- V1. For every process p such that $proc\_dom(p) = d$ and every file f, $READ\_OUT(p, f)(s) = READ\_OUT(p, f)(t)$, i.e. output consistency from the READ statement. Or equivalently, if $R \in acc\_pol(d, type(f))$ then both of the following hold:
  - $p \in in\_use\_set(s)(f)$ if and only if $p \in in\_use\_set(t)(f)$, and
  - if $p \in in\_use\_set(s)(f)$, then $data(s)(f) = data(t)(f)$.
- V2. For every process p such that $proc\_dom(p) = d$ and every file f, $TEST\_LOCK\_OUT(p, f)(s) = TEST\_LOCK\_OUT(p, f)(t)$, i.e. output consistency from TEST_LOCK statement. (For the initial definition of TEST_LOCK_OUT, this means if $W \in acc\_pol(d, file\_type(f))$, then $file\_lock(s)(f) = p$ if and only if $file\_lock(t)(f) = p$.) For the final version of TEST_LOCK_OUT, this means if $WR \in acc\_pol(d, file\_type(f))$, then $p = file\_lock(s)(f)$ if and only if $p = file\_lock(t)(f)$.
- R3. For every process p such that $proc\_dom(p) = d$ and every file f, $TEST\_OPEN\_OUT(p, f)(s) = TEST\_OPEN\_OUT(p, f)(t)$, i.e. output consistency from TEST_OPEN statement. Equivalently, if $R \in acc\_pol(d, file\_type(f))$, then $in\_use\_set(s)(f) = \phi$ if and only if $in\_use\_set(t)(f) = \phi$.

Note that R3 will be removed from the equivalence relation when TEST_OPEN is removed from the system. Also note that the condition for output consistency to hold for TEST_LOCK_OUT changes when its definition changes.

However, when the analysis is completed, it will be determined that two more conditions must be added to the equivalence relation for the system to be *step consistent* and to *locally respect* $\hookrightarrow$. These two conditions are:

- V3. For every file f such that $R \in acc\_pol(d, file\_type(f))$ and $file\_lock(s)(f) = file\_lock(t)(f)$ and no processes of domain d are in the in_use_set in both states s and t, $data(s)(f) = data(t)(f)$.
- V4. For every file f such that $R \in acc\_pol(d, file\_type(f))$ and no processes of domain d are in the in_use_set in both states s and t, $file\_lock(s)(f) = \phi$ iff $file\_lock(t)(f) = \phi$.

The reasons for these additions are explained below.

3.3 Initial State and Reachable States

We place the condition on the initial state that if a file in the system is locked, there can be no processes in the in_use_set of that file. Since the LOCK operation can only be successfully executed on a file with empty in_use_set and since the OPEN operation can only be successfully executed on a file which is not locked by any process, one can see by a simple induction argument that there can not exist a file which is both locked and its in_use_set is not empty.

3.4 Covert Channels Discovered

There were two covert channels discovered while trying to prove security for the system. Both of these covert channels led to changes in the operations.

3.4.1 TEST_OPEN and OPEN

A covert channel was found in the analysis of the operation OPEN. First, the analysis that led to the discovery of the covert channel will be discussed. Then, it will be shown how this channel was removed. The channel was found while trying to prove the locally respects $\hookrightarrow$ condition for the OPEN operation.

Let d and u be domains such that $d \not\rightsquigarrow u$. Let st be any state. Let p_0 be a process such that $d = proc\_dom(p_0)$. Let f_0 be a file. Consider the action $OPEN(p_0, f_0)(st)$, and let new_st be the state that results from the action.

Suppose that domain d and domain u both have read access to files of $file\_type(f_0)$. Further suppose that there are no processes in the in_use_set of f_0 and no processes have file f_0 locked in state st. Then in state new_st, $p_0 \in in\_use\_set(new\_st)(f_0)$. The problem is in the output consistency of TEST_OPEN_OUT, i.e. R3, because the output from TEST_OPEN in st is "F", and the output in new_st is "T". So $st \overset{y}{\not\sim} new\_st$.

This problem is a covert channel because domain d is supposed to be unable to "communicate" with domain u. However, by domain d having process p_0 open and close the file, and by domain u repeatedly checking TEST_OPEN a string of 1's and 0's can be sent from d to u. Thus there is a covert channel.

Once this channel was discovered, it was decided to remove the TEST_OPEN operation because it seems unnecessary. It can be inferred that the file is locked by simply using the READ operation. If data is output, then the process is in the in_use_set of the file. If no data is output, then the process is not in the in_use_set of the file. Moreover, now that the operation is removed, it is no longer necessary to have R3 as part of the equivalence relation, because output consistency from TEST_OPEN is not needed. So this covert channel has been averted.

3.4.2 TEST_LOCK and LOCK

A covert channel was also found in the analysis of LOCK. However, in this case, the fact that it is a covert channel is more subtle. It was discovered through the analysis of TEST_LOCK'S output consistency in showing that LOCK satisfies weakly step consistency.

Let d and u be domains. Let s and t be states such that $s \overset{d}{\sim} t$ and $s \overset{u}{\sim} t$. Let p_0 be a process such that $d = proc\_dom(p_0)$. Let f_0 be a file. Consider the actions $LOCK(p_0, f_0)(s)$ and $LOCK(p_0, f_0)(t)$, and let new_s and new_t, respectively, be the states that results from the actions.

Suppose that domain d has write access to files of $file\_type(f_0)$. Further suppose that there are no processes in the in_use_set of f_0 in state s, but there are processes, none of which are of domain d, in the in_use_set of f_0 in state t. Also assume that no processes have file f_0 locked in state s and state t. Then in state new_s, $p_0 = file\_lock(new\_s)(f_0)$ because LOCK can be executed. However, in new_t, $\phi = file\_lock(new\_t)(f_0)$, because LOCK can not be executed due to the in_use_set not being empty. The problem is in the output consistency of TEST_LOCK_OUT, i.e. V2 (recall we are currently using the old definition), when $d = u$. The output from $TEST\_LOCK\_OUT(p_0, f_0)(new\_s)$ is "T", and the result from $TEST\_LOCK\_OUT(p_0, f_0)(new\_t)$ is "F". Therefore, $new\_s \overset{y}{\not\sim} new\_t$. This failing is of the weakly step condition when $d = u$.

This failing is a covert channel. Consider the following scenario. There is a domain u such that $u \not\rightsquigarrow d$ and a state $st1$. Suppose further that u has read access to files of type $file\_type(f_0)$ and that $in\_use\_set(st1)(f_0) = p$, where p is a process of domain u. Then suppose the following sequence of actions are done: 1. $LOCK(p_0, f_0)(st1)$ which does not allow p_0 to lock the file because the in_use_set is not empty, and keeps the state $st1$, 2. $TEST\_LOCK\_OUT(p_0, f_0)(st1)$, which outputs a "F" to domain d and keeps the state $st1$. Then suppose $CLOSE(p, f_0)(st1)$ is done and $st2$ is the resulting state. Then $in\_use\_set(st1)(f_0) = \phi$, and by the locally respects $\hookrightarrow$ condition, we know that $st1 \overset{d}{\sim} st2$. However if the same sequence of actions is run on $st2$, this time domain d will get an output of "T" because process p_0 will have successfully locked the file. Then

domain d could do $CLOSE(p_0, f_0)(st2)$ to return to state $st1$ and the process would start again. So by domain d interpreting the "T" and "F" as 1's and 0's, it could receive information from domain u. This is the covert channel.

To fix this covert channel, TEST_LOCK_OUT is changed so that only processes with both read and write access to the file can get output from a TEST_LOCK. This change is satisfactory, because it is assumed that usually a process is writing to a file that it can also read. It is assumed to be an unusual case when a process is writing to a file that it can not read. Notice that this change will also affect the equivalence relation, because output consistency for TEST_LOCK now has slightly different requirement.

These are the only covert channels found in this system. The other failings of the system were not covert channels. Instead those problems were fixed with changes and additions to the equivalence relation.

3.5 Equivalence Relation Changes

There were three places where the equivalence relation had to be changed. Below explains why these changes were needed.

3.5.1 Initial Version of V3

Weakly step consistent will fail for the operation OPEN. Let d and u be domains. Let s and t be states such that $s \overset{d}{\sim} t$ and $s \overset{u}{\sim} t$. Let p_0 be a process such that $d = proc\_dom(p_0)$. Let f_0 be a file. Consider the actions $OPEN(p_0, f_0)(s)$ and $OPEN(p_0, f_0)(t)$, and let new_s and new_t, respectively, be the states that results from the actions.

Consider the case when $u = d$ and domain d has read access to $file\_type(f_0)$. Suppose there are not any such processes of domain u in the in_use_set of file f_0 in both state s and in state t. Further suppose that $data(s)(f_0) \neq data(t)(f_0)$. Then if $OPEN(p_0, f_0)$ is acted upon each state, followed by $READ(p_0, f_0)$, different values will be output from state s and state t. So V1 fails when trying to prove the equivalence of the new states. The minimal fix for this turns out to be V3 with the one weakening that $file\_lock(s)(f) = file\_lock(t)(f) = \phi$ instead of $file\_lock(s)(f) = file\_lock(t)(f)$.

3.5.2 Change V3

The final version of V3 is created to fix a problem with UNLOCK. The failing occurs in the analysis of weakly step consistence holds for UNLOCK. Let d and u be domains. Let s and t be states such that $s \overset{d}{\sim} t$ and $s \overset{u}{\sim} t$. Let p_0 be a process such that $d = proc\_dom(p_0)$. Let f_0 be a file. Consider the actions $UNLOCK(p_0, f_0)(s)$ and $UNLOCK(p_0, f_0)(t)$, and let new_s and new_t, respectively, be the states that results from the actions.

Assume $R \in acc\_pol(u, type(f_0))$ and suppose that $data(s)(f_0) \neq data(t)(f_0)$. Further suppose that $p_0 = file\_lock(s)(f_0) = file\_lock(t)(f_0)$. Then when

$UNLOCK(p_0, f_0)$ is acted on both states s and t, the resulting states new_s and new_t violate the V3 part of the u-equivalence of new_s and new_t. To fix this problem, the condition $file\_lock(s)(f) = file\_lock(t)(f) = \phi$ is strengthened to $file\_lock(s)(f) = file\_lock(t)(f)$.

3.5.3 Add V4

Another problem arises in the analysis of weakly step consistence of the OPEN operation. Let d and u be domains. Let s and t be states such that $s \overset{d}{\sim} t$ and $s \overset{u}{\sim} t$. Let p_0 be a process such that $d = proc\_dom(p_0)$. Let f_0 be a file. Consider the actions $UNLOCK(p_0, f_0)(s)$ and $UNLOCK(p_0, f_0)(t)$, and let new_s and new_t, respectively, be the states that results from the actions.

Suppose $d = u$. Consider the case where $file\_lock(s)(f_0) = \phi$ and $file\_lock(t)(f_0) \neq \phi$. Assume that $R \in acc\_pol(d, file\_type(f_0))$. Then $p_0 \in in\_use\_set(new\_s)(f_0)$ because the OPEN operation is successful, but $p_0 \notin in\_use\_set(new\_t)(f_0)$ because the non-empty $file\_lock(t)(f_)$ prevents the OPEN operation. However, this means that a READ operation on f_0 from process p_0 will give $data(s)(f_0)$ as output, but READ operations on f_0 in state new_t will not give output. This contradicts V1 of $s \overset{u}{\sim} t$. The minimal resolution of this problem is the addition of V4 to the equivalence relation.

These are all the changes to the equivalence relations that needed to be implemented so that the Unwinding Theorem for Intransitive Policies can determine that the system is secure. Notice that no assumptions need to be added. Once the system and the equivalence relation are defined as above, the Unwinding Theorem for Intransitive Policies shows that the system is secure.

Chapter 4

Comparison with the Algorithm for Transitive Systems

Though our analysis for the intransitive system paralleled that of the transitive system in certain respects, there were also a number of distinct differences. In both cases, a channel in the system was discovered by examining the OPEN request; a process p_1 which has read access to a certain file but does not have write access to that file can transmit information to another process p_2 in a different security level or domain which also has read access to the file by opening and closing it while p_2 executes TESTOPEN. It is appropriate to remove TESTOPEN from the system in both cases. It was also necessary to place similar restrictions in both systems on the situations in which the data in a file needs to be identical in two different states in order for those states to be considered equivalent.

However, we occasionally needed to implement a different system modification than in the transitive system simply because of the unordered nature of the intransitive system. Specifically, sometimes a covert channel was found in both systems that could easily be fixed in the transitive system by adding a requirement which involved a process executing "at the same level as a file" in the system. Since an intransitive system does not have the ordering concept of "at the same level as a file", fixing such a channel involves finding an effective and reasonable requirement on process domain access permissions alone to eliminate the channel. However, we were always able to find such a requirement in our analysis. For instance, the TESTLOCK operation allows a channel in the transitive system if a process p_1, executing at some security level, is able to lock and unlock a file while a second process p_2, executing at a lower level, tests to see if the file is locked. This channel is easily fixed in the transitive system by requiring that a TESTLOCK request can only be executed on a file which is at the same level as the process executing it. In our intransitive system, the same

type of problem occurs a process with write access to a file is able to lock and unlock the file, while another process with write access but not read access can test to see if the file is locked or not. We are able to eliminate this channel by modifying TESTLOCK so that both read and write access to a file are required for a process in order to execute the TESTLOCK operation.

Finally, by decision we took a different approach toward reachable states in the algorithm with our intransitive system, which resulted in several changes that were not analogous to those in the transitive system. The analysis for the transitive system does not place any restrictions on initial states, and therefore must consider cases where an undesirable initial state allows a reachable state that permits an obvious covert channel. Restrictions are then added to A which deal with these states. Our approach was to place restrictions on initial state before starting the algorithm in order to eliminate certain clearly undesirable states from consideration. For example, we did not allow the system to begin in a state where a file can be locked and still have processes in its *in_use_set*. This led to a different but perhaps more efficient choice of system modifications in order to eliminate similar covert channels. One possible covert channel in both original systems occurs when a process with read access to a file is allowed to open and close the file while another process with write access but not read access tries to lock the file and uses TESTLOCK to determine if the request was successful. The analysis for the transitive system modifies the LOCK operation to allow processes in a file's *in_use_set* when a file is locked; however, it retains the property that the TESTLOCK operation can only return a true or false value to a process, and therefore a process can never distinguish whether it has locked a file or not. In our intransitive system, we retain the requirement that a file cannot be locked while it has processes in its *in_use_set*. We then allow TESTLOCK to return true only if the executing process was the process having the file locked, and use the fact from a previous modification that a process must have read access to a file in order to execute TESTLOCK to eliminate the channel.

Chapter 5

Conclusion

We successfully proved for our system that the conditions for the unwinding theorem held. Also, we have shown the algorithm to be a reasonable approach for analysis. In the analysis, we assumed domains and types were fixed. For future research, redefining the system to include domain and type changes may pose interesting questions in both formulation and analysis of intransitive non-interference security policies. Using PVS, we verified the security of the system in some cases.

Bibliography

[1] TODD FINE, *Constructively Using Noninterference to Analyze Systems*, IEEE Symposium on Security and Privacy, May, 1990, pp. 162–169 .

[2] JOHN RUSHBY, *Noninterference, Transitivity, and Channel-Control Security Policies*, SRI International, Dec, 1992.