Information Technology Division

Research Report
ERL-0577-RR

The Application of Higher Order Logic to Security Models.

by

A. Cant and K. Eastaughffe

DSTO AUSTRALIA

Electronics Research Laboratory

Information Technology Division

Research Report
ERL-0577-RR

The Application of Higher Order Logic to Security Models.

by

A. Cant and K. Eastaughffe

SUMMARY

This paper describes the application of the proof assistant HOL (Higher Order Logic) to reasoning about security models. Using Rushby's general framework for security models, we show how the HOL system can prove an unwinding theorem for non-interference of processes at different security levels. The method of unwinding is then applied to the Low Water Mark Model of security. From this analysis, we draw conclusions about the strengths and weaknesses of HOL as a reasoning tool.

NOV 91

COPY NO 89

POSTAL ADDRESS: Director, Electronics Research Laboratory,PO Box 1500, Salisbury, South Australia, 5108

CONTENTS

Page No.

LIST OF TABLES

Page No.

1 Introduction

Formal methods are becoming more and more widely used in computer science and software development. Specification languages, automatic theorem provers, verification tools and other systems based on formal mathematical techniques all have a crucial role to play. The need for formal methods is especially acute in systems which are safety or security-critical. In such cases one failure could be disastrous, leading to loss of life, damage to the environment, breaches of national security etc. In such cases we wish to have the assurance that the software will function correctly.

We need to be clear on the terminology. Software can be termed *correct* only with respect to a formal detailed specification which describes exactly what the program is required to do. The objective of *program verification* is to prove mathematically that every execution of the program will satisfy the given specifications.

A hierarchical approach to system design is now a well-established technique. Such an approach enables the detailed formal specifications themselves (which can be complex and subject to logical and other errors) to be proved correct (verified) against a simpler, more abstract specification. In this case, we speak of *design verification*. Again, this specification may itself need to be verified against an even simpler and yet more abstract specification, until we eventually reach the formal top-level specification. The specifications reflect different levels of detail in the design.

System specifications are naturally expressed in a mathematical notation, and the process of design verification uses techniques of mathematical proof: we construct a mathematical model of the system by formulating a theory based on certain axioms, and proving theorems from these axioms. The mathematician may be content with — and convinced by — paper and pencil proofs. However, mathematical models in the safety and security critical world need to be subjected to a greater level of rigour. In such cases an automatic reasoning tool can help us formulate the theory, manage the proofs of key results and avoid logical errors, leading to increased assurance of correctness.

This paper is concerned with the application of one such reasoning tool, namely the Higher Order Logic (HOL) system (developed by Professor Mike Gordon at University of Cambridge [1]) to the area of *security models*.

HOL was originally suggested as a tool for the verification of hardware [2]. Although most of the activity in HOL is still in this area, HOL has also been applied to protocol verification [3], mathematical theories such as groups and integers [4], machine architecture specification [5], and, most recently, to program verification [6]. Until recently, HOL has not been applied to security policy modelling [7]. We believe that there will be increased interest in this area once tools for the study of concurrency with HOL become available [8]).

In order to verify that a system satisfies some notion of security, the system needs to be modelled mathematically and a mathematical definition of security needs to be formulated. The definition of security varies depending on the security needs of the users of the system. We have followed the development of Rushby's general framework for security models [9] which defines security in terms of non-interference — a system is secure if certain processes do not interfere with certain other processes, i.e. they do not affect the other processes' output and view of the system.

As an instance of this general theory we shall give a detailed treatment of the Low Water Mark Model for security. Although not a realistic model of security because of its simplicity, the Low Water Mark Model has some interesting features. It has been studied by a number of authors [10, 11, 12], as a case study for a range of automatic verification and theorem proving tools.

The purpose of this paper is not to break new ground in the theory of security models, but rather to show how effective HOL can be in a new problem domain such as security. We shall see that HOL can give useful insights into this field. The use of HOL in a new area also helps us to highlight HOL's strengths and weaknesses, and to see how it should be extended or modified to become more useful in other application areas.

In the next section we shall give a description of a general framework for security models, based on [9]. This is followed by a brief description of the HOL system. In subsequent sections we show how HOL can be used to prove Rushby's general unwinding theorem for non-interference of processes, and

consider in detail the case of the Low Water Mark model. The interaction with HOL is described in some detail, in the hope that readers who have no experience of HOL will be able to appreciate how a medium-sized proof is tackled.

2 Security Models

In terms of the hierarchical approach referred to in the Introduction, the verification of security requirements requires that the top level specification be the assertion that the system is secure with respect to some definition of security. This definition of security will usually require a description of a general model of a system. This will involve the identification of the types of the system entities and some abstract functions whose domains and ranges are in the identified types. These abstract functions are then used to define security.

The next level of specification, describing a particular instance of the general system, will be more detailed, including the declaration of particular operations and a description of their effect on the system. Also, the structure of certain entities may be further decomposed, and the functions which were abstract at the previous level may be given concrete definitions. It is the verification of this second level specification against a definition of security on which we have focused.

2.1 A General Model

The security model of a system can be formalized in many ways. There is presently much debate on what an appropriate mathematical formulation of security should be. Two important concepts are *deducibility security* [13] and *restrictiveness* [14] addressing such issues as non-deterministic systems and for the latter, under what conditions secure systems remain secure when hooked-up together. Restrictiveness is currently being modelled in HOL by Levitt and Alves-Foss [7], who have proved that restrictivenes satisfies hook-up security and shown that a simple distributed system satisfies restrictiveness. In our work, we have studied security in terms of *non-interference*, using the state-based description which has been formulated by Rushby [9], based on the earlier work of Goguen and Meseguer [15]. A system consists of the following:

- a set S of *states*, with an initial state called *initstate*;
- a set P of *processes*[1];
- a set C of *commands*; and
- a set O of *outputs*.

We also have functions

$$next : S \times P \times C \rightarrow S$$
$$out : S \times P \times C \rightarrow O$$

Here $next(s, p, c)$ denotes the state of the system obtained when the process p performs the command c in state s; while $out(s, p, c)$ denotes the output returned by that command. Elements of A = P × C will be called *actions*. We are especially interested in the set A^* of action sequences (here regarded as lists, for convenience). The function next can be extended to the function

$$nextlist : S \times A^* \rightarrow S$$

by defining

$$nextlist(s, []) = s, and$$
$$nextlist(s, h :: t) = next(nextlist(s, t), h)$$

where [] denotes the empty list, and $h::t$ denotes the list with head h and tail t. We also define the functions

$$do : A^* \rightarrow S$$
$$result : A^* \times A \rightarrow O$$

[1] For the sake of generality we use *processes* as the active agents rather than *users* as used by Rushby.

by the equations

$$do(\alpha) = nextlist(initstate, \alpha),$$
$$result(\alpha, p, c) = out(do(\alpha), p, c)$$

Thus $do(\alpha)$ yields the state reached after performing the action sequence α, while $result(\alpha, p, c)$ gives the output when process p does command c after action sequence α.

As a result of the outputs of some sequence of actions performed starting from the initial state of the system, a process will have a *view* of each of the states that have been reached. Formally, we have a function

$$view : P \times S \rightarrow \overline{S}$$

where $\overline{S}$ is some (as yet unspecified) set of *private states*. At the top level of specification it is possible to give a formal definition of security without defining what states, outputs and views look like or how a state is changed by an action.

A process p_1 is said to be *non-interfering* with another process p_2 if the results seen by the process p_2 after any sequence of actions is the same regardless of whether process p_1 performed some of the actions or not. Formally:

$$result(\alpha, p_2, c) = result(\alpha/p_1, p_2, c) \quad \forall \alpha \in A^*, \ c \in C$$

where α/p denotes the sublist of α formed by deleting all actions consisting of commands performed by the process p.

A definition of security can be formulated by firstly declaring some relation R on processes (called a *security policy* by Rushby), where $R\ (p_1, p_2)$ is interpreted to mean that information is allowed to flow from process p_2 to process p_1. We shall say that the system is *secure* with respect to a policy R if p_2 is non-interfering with p_1 whenever $R\ (p_1, p_2)$ is not true. In other words, if no information is allowed to flow from p_2 to p_1, then p_1 should ne unaffected by whatever p_2 does.

At this level of detail it is not possible to prove security with respect to non-interference because the concept of a view has not been formally defined. However, it is possible to prove certain theorems, including an unwinding theorem which states that if a given definition of the view of a process satisfies certain conditions then the system is secure. These conditions are intended to be simpler to prove and thus, once the system is described in more detail at the second level of specification, only these conditions need to be proved rather than proving security from first principles.

Before we give the formal statement of the unwinding theorem, we make the following definition. We say that the system is *internally consistent* if

$$view(p, s) = view(p, t) \Rightarrow out(s, p, c) = out(t, p, c) \quad \forall p \in P, \ s, t \in S, \ c \in C,$$

which expresses the fact that, if a process p's view of two states s and t is the same, then the outputs of commands performed by p will always be identical for these states.

The unwinding theorem is as follows [9]:

Theorem: Let M be an internally consistent system with a security policy R such that

$$(1)\ \ R(p_1, p_2) \Rightarrow view(p_2, next(s, p_1, c)) = view(p_2, s)$$
$$(2)\ \ view(p_1, s) = view(p_1, t) \Rightarrow view(p_1, next(s, p_2, c)) = view(p_1, next(t, p_2, c))$$
$$\forall p_1, p_2 \in P, \ \ s \in S, \ \ c \in C$$

Then M is secure (with respect to the policy R).

The proof of this theorem is straightforward, and may be found in [9].

It is useful to prove this unwinding theorem at the most abstract level possible, because it can then be applied to many different descriptions of systems which may have different definitions for states, operations, outputs and views but which, nevertheless, satisfy the sufficiency conditions for security according to the unwinding theorem. There are many different versions of unwinding theorems but, as hinted by their name, they reduce the problem of security from proving certain assertions about sequences of actions to proving assertions about single actions.

In Section 4 the top level model of the system is set up within HOL and the unwinding theorem is proved.

The next level of specification describes the system in more detail by giving definitions to states, operations, actions (and their effect on states), outputs of actions, views, and a security policy. This description can still be general in that many real systems may satisfy the definitions; but, because it is less abstract than the top level description, it characterises a proper subset of the set of possible systems described by the top level model. One such lower level model is the Low Water Mark Model.

2.2 The Low Water Mark Model

This model describes a system consisting of a set of processes, a non-empty set of data objects and three operations used by the processes to access the data objects. Associated with each data object is a classification which is a member of set of values called *levels*. Associated with each process is a clearance level which is also a member of this set. On this set of levels there exists a relation which will be called **dominates**, where "dominates(x,y)" means that x has a higher security level than y. It is assumed that this relation is a partial ordering, i.e. it is reflexive, antisymmetric and transitive. We shall assume that there is a special level, called **system high**, which dominates all other levels. A state can be described as a mapping from data object identifiers to the pair consisting of the contents of the data object and the classification of the object.

The three operations are *reading* the contents of a data object, *overwriting* the contents of a data object, and *resetting* the classification of an object to **system high**. The effect and output of an operation on a data object performed by a process is dependent on the relationship between the classification of the object and the clearance of the process. The effect and output of each of the three operations paired with a process are:

1. READ
   The state of the system remains the same after a process has read the contents of an object. If the clearance of the process dominates the classification of the object, then the output signals that the read was successful and the contents of the object are displayed. Otherwise, the output signals that the read was unsuccessful and no other information is displayed.

2. WRITE
   If the classification of the object dominates the clearance of the process then the new state after a write remains the same, except that the identifier of the object which was overwritten is now mapped to the pair consisting of the new data and the clearance of the process. Otherwise, the state remains the same and the output signals an unsuccessful write.

3. RESET
   If the classification of the object dominates the clearance of the process then the new state after a reset remains the same, except that the data object's contents are cleared and its classification is now set to **system high**. Otherwise, the state remains the same and the output signals an unsuccessful reset.

The model is called the Low Water Mark Model because a data object's classification can only be increased by a reset operation, which will set it to **system high**.

The unwinding theorem can be used to prove the security of the Low Water Mark Model given the additional assumption that the ordering **dominates** is a *total* relation, that is, given any two levels, one must dominate the other. An unwinding theorem specific to this model was derived by Billard in [16].

Before formulating the above theory in HOL, we shall give a brief overview of the HOL system.

3 The HOL System

The HOL system is large and complex, and it is beyond the scope of this paper to describe it fully. We refer the reader to the HOL manuals [4] for more details. In the following, we shall attempt to give a flavour of working with HOL, as well as highlighting those features of the system relevant for reasoning about security models.

HOL supports interactive theorem proving in higher order logic. It provides a natural and highly expressive way of specifying and reasoning about models of abstract systems — as well as mathematical theories in general. It inherits many ideas from the earlier LCF theorem prover developed by Robin Milner and collaborators in the early 1970's [17]. As in LCF, the programming language ML (ML stands for Meta-Language) provides the environment in which terms and theorems of the logic are denoted and theorem proving takes place. We shall assume that the reader has some familiarity with ML (see [18] for an elementary introduction).

HOL is really a proof-assistant and proof checker. It will not prove complex theorems automatically; the user must have an idea of the way the proof will work, and apply the appropriate steps (called tactics) in the proof, which proceeds in a goal-directed fashion. The HOL system manages the proof, taking care of the details of primitive proof steps, and provides a sound theorem proving environment — i.e. the user is assured that a theorem, once obtained, is true within the logic.

3.1 The HOL Logic

The HOL logic is a version of *higher order logic* based on Church's formulation of simple type theory [19]. It is a variant of typed polymorphic λ–calculus, with formulae being identified with terms of boolean type. Variables can range over functions and predicates, and functions can take other functions as arguments (hence 'higher order').

Terms of the HOL logic have the ML type called `term`, and are input to HOL enclosed in quotation marks. The following table, adapted from [4], summarises the primitive terms of the logic:

Table 1 Primitive Terms of the HOL Logic

Kind of term	HOL Notation	Description
Variable	"var : σ "	variable var of type σ
Constant	"const : σ "	constant of type σ
Combination	"t t' "	function t applied to t'
Abstraction	"\x. t "	lambda expression

From these primitive terms are built the usual logical constructs, as follows:

Table 2 Derived Logical Constructs of HOL

Kind of term	HOL Notation	Description
Truth	"T "	true
Falsity	"F "	false

Table 2 (Continued) Derived Logical Constructs of HOL

Kind of term	HOL Notation	Description
Negation	"¬t "	not t
Disjunction	"t ∨ t' "	t or t'
Conjunction	"t ∧ t' "	t and t'
Implication	"t ==> t' "	t implies t'
Equality	"t = t' "	t equals t'
Universal Quantification	"!x.t "	for all x : t
Existential Quantification	"?x.t "	there exists an x such that t
Unique Existential Quantification	"?!x.t"	there exists a unique x such that t
ϵ-term	"@x.t "	an x such that t
Conditional	"t => t' | t'' "	if t then t' else t''

The types of HOL terms have the ML type called `type`, and can take the following forms:

Table 3 HOL Types

Kind of Type	HOL Notation	Description
Type variable	":*α"	arbitrary type
Type constant	":bool "	fixed type
Function type	":σ -> σ' "	functions from σ to σ'
Compound type	":(σ_1, ..., σ_n) op"	general type constructor

Any term input to the system must be well-typed according to the rules of the logic. HOL has a type checker for logical terms based on the ML type checking algorithm.

3.2 Proving Theorems in HOL

While interacting with the HOL system the user is working with an object called a *theory*. A theory consists of types, constants, definitions and axioms. It also contains an explicit list of those theorems which have so far been proved. A theorem is represented in HOL by a value of ML type `thm`. The system is *sound* in that the only way to obtain theorems is by generating a proof. This is done by applying ML functions representing inference rules, either to axioms or previously generated theorems. Theorems are denoted generally by $\Gamma \vdash t$, where Γ is a set of boolean terms called *assumptions*, and t is a boolean term called the *conclusion*. If Γ is empty, we write simply $\vdash t$.

The HOL logic itself has five axioms and eight primitive inference rules, along with a vast number of derived theorems and inference rules. The HOL system is provided with a number of built-in theories (such as **bool**, **pair** and **list**), as well as a set of useful library theories (such as **sets**, **string**, **integer** etc) which can be called upon at will.

In practice, proofs are not carried out forwards, but in a more natural goal-directed fashion invented by Robin Milner for LCF. Milner invented the notion of *tactics*. A tactic is an ML function which

1. reduces a goal to subgoals, and
2. remembers the reason why solving the subgoals will solve the goal.

For example, suppose we want to prove the formula A ∧ B. Then the goal-directed approach says that it is enough to prove the subgoals A and B, because we know that from ⊢ A and ⊢ B we can deduce the theorem ⊢ A ∧ B.

HOL has an extensive subgoal package for managing goal-directed proofs. Becoming a HOL expert means becoming familiar with a range of tactics, and the situations where they can be applied. New tactics can be programmed in ML, or (more easily) built up from existing tactics by means of special functions called *tacticals*. Quite powerful and sophisticated tactics (which we might term *strategies*) can be tailor-made for the problem domain being studied.

4 A Theory of Security in HOL

In this section we shall show how to formalise within HOL the general theory of security which was described in Section 2. We describe, at the top level of abstraction, the specification of non-interference and security, and the interactive proof of the unwinding theorem in HOL.

4.1 Type and Constant Definitions

We begin the interaction with HOL by creating a new HOL theory called 'security'. Within this theory all the entities, functions and relations for the general system are expressed formally in the higher order language of HOL. Firstly, all the necessary types need to be declared by means of the following ML commands (the symbol # is the HOL prompt, and the HOL responses are suppressed).

```
#new_theory 'security';;
#new_type 0 'process';;
#new_type 0 'command';;
#new_type 0 'state';;
#new_type 0 'output';;
#new_type 0 'private_state';;
#new_type_abbrev ('action',":process#command");;
#new_type_abbrev ('policy',":process -> process -> bool");;
```

The first five types are declared as abstract types because, at this level of detail, there is no further information on the nature of the objects in these sets. However, we do know that an action is a pair consisting of a process and a command, and that a policy is a relation on processes. Therefore, these types can be defined in terms of the abstract types.

We also need to declare the primitive constants of the system (note that, in higher order logic, functions are also constants). The values of the `view` function are taken to be of the type `private_state`.

```
#new_constant ('initstate', ":state");;
#new_constant ('next', ":state -> action -> state");;
#new_constant ('out', ":state -> action -> output");;
#new_constant ('view',":process -> state -> private_state");;
```

The functions defined in Section 2 are now easily expressed in HOL. The following interaction with HOL shows how `nextlist` is defined in ML, and also HOL's 'reply' once the definition is accepted. Note that `nextlist` is defined recursively on action lists. Proofs involving recursively defined functions will typically involve induction on the recursive type.

```
let NEXTLIST_DEF = new_list_rec_definition ('NEXTLIST_DEF',
   "(nextlist (s:state) ([]:(action)list) = s) /\
    (nextlist s (CONS h t) = next (nextlist s t) h)");;
```

```
NEXTLIST_DEF =
  |- !s. nextlist s[] = s
     !s h t. nextlist s(CONS h t) = next nextlist s t h
```

To make what follows more readable, from here on we shall only show the HOL responses to new user definitions. The functions do and `result` are defined as follows:

```
DO_DEF = |- !alist. do alist = nextlist initstate alist

RESULT_DEF = |- !alist a. result alist a = out(do alist)a
```

Since the definition of non-interference involves a comparison of two lists of actions — one of which is a sublist of the other — we define a function `filter` which, given the original action list and process, will return the appropriate sublist. It is then possible to define non-interference concisely.

```
FILTER_DEF =
  |- !p. filter[]p = []
     !a alist p.
       filter(CONS a alist)p =
         FST a = p => CONS a(filter alist p) | filter alist p

NON_INTERFERENCE_DEF =
  |- !p1 p2.
     non_interferes p1 p2 =
       !alist p2'.
         result(filter alist p1)p2' =
         result(filter alist p1)p2'
```

We are now in a position to define security with respect to non-interference for an arbitrary security policy ***R***:

```
SECURE_DEF =
  |- !R. secure R = !p1 p2. R p1 p2 ==> non_interferes p1 p2
```

The definition of internal consistency is as follows:

```
INTERNALLY_CONSISTENT_DEF =
  |- internally_consistent =
     !s t p. view p s = view p t ==>
       !c. out s(p,c) = out t(p,c)))
```

This completes the specification of the general system for security. As can be seen, it is quite easy to formalise it in HOL. There are a number of reasons for this. Firstly, the higher order capability of HOL means that we have no difficulty in defining concepts such as security, where we need to quantify over a predicate R (the security policy). Secondly, the fact that the HOL logic is strongly typed means that many trivial errors in setting up the specification can be avoided. Thirdly, it is easy to define new functions on recursive types such as `nextlist`.

4.2 An Interactive Proof

Having set up the basic theory, we can now prove the unwinding theorem using HOL's subgoal package, which enables a user to develop a goal-directed proof interactively by issuing commands (tactics) which produce new subgoals. The package manages the proof by recording which subgoals still need to be proved in order to prove the original goal.

To carry out the proof, we use a lemma called UNAFFECTED_VIEW_LEMMA:

```
UNAFFECTED_VIEW_LEMMA =
|- !R.
   internally_consistent /\
     !p1 p2.
      R p1 p2 ==>
         !alist.view p2(do(filter alist p1)) = view p2 do alist
   ==> secure R
```

This lemma asserts that, if information flow from process p1 to process p2 is not allowed according to a policy R, then p2's view should not be affected by commands performed by p1. The proof of this lemma is short and straightforward, and is included in Appendix A.

The proof of the unwinding theorem begins by setting the top goal to the HOL formulation of the theorem using the definitions above.

```
#g "!R. R:policy. internally_consistent /\
       !p1 p2.
         R p1 p2 ==>
           !s c. view p2 (next s (p1,c)) = view p2 s) /\
       ! p1 s t. (view p1 s = view p1 t) ==>
           !c p2. (view p1 (next s (p2,c)) =
                          view p1 (next t (p2,c)))
    ==> secure R";;

"!R.
  internally_consistent /\
  !p1 p2.
    R p1 p2 ==>
     !s c. view p2 (next s(p1,c)) = view p2 s) /\
     !p1 s t.
      (view p1 s = view p1 t) ==>
      (!c p2. view p1(next s(p2,c)) =
         view p1(next t(p2,c))))) ==> secure R"
```

Firstly, the goal is decomposed to its simplest form which requires `secure R` to be proved with the three conditions as assumptions. For this we use the standard tactic `REPEAT(STRIP_TAC)`, which breaks apart conjunctions and implications, and removes outermost quantifiers. We can then use the above lemma, which has as its consequence `secure R`. Therefore, if we can show that the conditions in UNAFFECTED_VIEW_LEMMA can be proved, we shall have solved the goal. These conditions of the lemma replace the goal if we use the tactic MATCH_MP_TAC.

These proof steps are combined into one tactic by using the tactical THEN. In general, *tac1* THEN *tac2* will apply *tac1* to the current subgoal and then apply *tac2* to all resulting subgoals.

In the following interactions with HOL, the current assumptions are listed below the goal and each one is delimited by square brackets. Assumptions are numbered from one, starting at the bottom of the list as displayed.

```
#e(REPEAT STRIP_TAC THEN
              MATCH_MP_TAC UNAFFECTED_VIEW_LEMMA);;
```

```
OK..
"internally_consistent =
 !p1 p2.
   R p1 p2 ==>
        !alist. view p2(do(filter alist p1))
               = view p2(do alist))"
    [ "internally_consistent" ]
    [ "!p1 p2. R p1 p2 ==>
        (!s c. view p2(next s(p1,c)) = view p2 s)" ]
    [  !p1 s t.
        (view p1 s = view p1 t) ==>
        (!c p2. view p1(next s(p2,c))
                        = view p1(next t(p2,c))))" ]
```

The conjunct `internally_consistent` can be solved simply by rewriting with the assumptions, using the HOL tactic `ASM_REWRITE_TAC`. In order to prove the other conjunct, our first impulse might be to apply `REPEAT STRIP_TAC`. However, we wish to apply the induction principle to the action list `al`, and for this the goal needs to be universally quantified over this term. `REPEAT STRIP_TAC` would strip away too much; instead the two outer quantifications are stripped using `GEN_TAC` and the implication is decomposed in two steps.

```
#e ASM_REWRITE_TAC[] THEN
             REPEAT GEN_TAC THEN
             STRIP_TAC );;
OK..
"!alist. view p2(do(filter alist p1))
         = view p2(do alist)"
    [ "internally_consistent" ]
    [ "!p1 p2.
        R p1 p2 ==>
        (!s c. view p2(next s(p1,c)) = view p2 s)" ]
    [ !p1 s t.
        (view p1 s = view p1 t) ==>
        (!c p2. view p1(next s(p2,c))
                        = view p1(next t(p2,c))))" ]
    [ "R p1 p2" ]
```

The term is now in the form where we can apply the induction tactic for lists, producing two subgoals — one for the base case of an empty list and one for the inductive step.

```
#e(LIST_INDUCT_TAC);;
OK..
2 subgoals
"!h. view p2(do(filter(CONS h alist)p1)) =
                          view p2(do(CONS h alist))"
    [ "internally_consistent" ]
    [ "!p1 p2.
        R p1 p2 ==>
        (!s c. view p2(next s(p1,c)) = view p2 s)" ]
    [ (!p1 s t.
        (view p1 s = view p1 t) ==>
        (!c p2. view p1(next s(p2,c))
                  = view p1(next t(p2,c))))" ]
    [ "R p1 p2" ]
    [ "view p2(do(filter alist p1)) = view p2(do alist)" ]
```

```
"view p2 (do(filter[]p1)) = view p2 do[])"
    [ "internally_consistent" ]
    [ "!p1 p2.
        R p1 p2 ==>
        !s c. view p2(next s p1,c)) = view p2 s " ]
    [ (!p1 s t.
        (view p1 s = view p1 t) ==>
        !c p2. view p1(next s(p2,c))
                = view p1(next t(p2,c)))" ]
    [ "R p1 p2" ]
```

The subgoal for the base case is easily solved by rewriting with the definition of `filter` since `filter [] p1 = []`. To make progress on the subgoal for the inductive step, it can also be rewritten with the definition of `filter`:

```
#e(REWRITE_TAC[FILTER_DEF]);;
OK..
"!h.
  view p2(do(filter alist p1)) =>
        filter alist p1 ! CONS h(filter alist p1))) =
  view p2(do(CONS h al))"
    [ "internally_consistent" ]
    [ "!p1 p2.
        R p1 p2 ==>
        !s c. view p2 next s(p1,c)) = view p2 s)" ]
    [ !p1 s t.
        view p1 s = view p1 t  ==>
        !c p2. view p1 next s(p2,c))
                    = view p1(next t(p2,c))) " ]
    [ "R p1 p2" ]
    [ "view p2(do(filter alist p1))
                                = view p2(do alist)" ]
```

This term can be simplified by stripping the outer quantifier and removing the conditional expression by performing case analysis on `FST h = p1`. It is possible to expand the two resulting subgoals by rewriting with the definitions of do and `nextlist`. This will remove occurrences of CONS and introduce occurrences of `next` enabling the term to be matched later on with consequences of assumptions. The tactical THEN is very useful here, since the resulting subgoals are of the same form and can thus be simplified or solved by the same tactics.

```
#e(GEN_TAC THEN COND_CASES_TAC THEN
   REWRITE_TAC[DO_DEF;NEXTLIST_DEF]);;
OK..

2 subgoals

"view p2(next(nextlist initstate(filter alist p1))h) =
 view p2(next(nextlist initstate alist)h)"
    [ "internally_consistent" ]
    [ "!p1 p2.
        R p1 p2 ==>
        !s c. view p2(next s(p1,c)) = view p2 s)" ]
    [ (!p1 s t.
        (view p1 s = view p1 t) ==>
        (!c p2. view p1(next s(p2,c))
                    = view p1(next t(p2,c))))" ]
    [ "R p1 p2" ]
    [ "view p2(do(filter alist p1))
                                = view p2(do alist)" ]
    [ "(FST h = p1) = F" ]
```

```
"view p2(nextlist initstate(filter alist p1)) =
 view p2 next nextlist initstate alist h."
    [ "internally_consistent" ]
    [ "!p1 p2.
        P p1 p2 ==>
        !s t. view p2(next s(p1,t)) = view p2 s)" ]
    [  !p1 s t.
        (view p1 s = view p1 t) ==>
        !c p2. view p1(next s(p2,c))
                        = view p1(next t(p2,c)))" ]
    [ "R p1 p2" ]
    [ "view p2(do(filter alist p1))
                                = view p2(do alist)" ]
    [ "(FST h = p1) = T" ]
```

At this point we shall concentrate on proving the first (i.e. bottom) subgoal. The proof of the second subgoal will follow the same steps. Now that the definition of `nextlist` has been used to remove occurrences of CONS, the goal is more readable if the terms of the form `nextlist initstate al` (where `al` is an action list) are rewritten in terms of do. We use our own inference rule SYM_GEN, which takes theorems of the form $\vdash$!x_1 ... x_n. t1 = t2 and gives the theorem $\vdash$!x_1 ... x_n. t2 = t1. The goal is also rewritten using the second assumption. We then use the powerful tactic RES_TAC, which searches for assumptions of the form A $\Rightarrow$ B and attempts to match A against other assumptions, using *modus ponens* to produce new assumptions matching B.

```
#e  ASM_REWRITE_TAC[SYM_GEN DO_DEF] THEN RES_TAC  ;;
OK..
"view p2 next(do alist)(h) = view p2(do alist)."
    [ "internally_consistent" ]
    [ "!p1 p2.
        P p1 p2 ==>
        !s t. view p2(next s(p1,t)) = view p2 s)" ]
    [ !p1 s t.
        view p1 s = view p1 t) ==>
        (!c p2. view p1(next s(p2,c))
                        = view p1(next t(p2,c))))" ]
    [ "R p1 p2" ]
    [ "view p2(do(filter alist p1))
                                = view p2(do alist)" ]
    [ "(FST h = p1) = T" ]
    [ "!s c. view p2(next s(p1,c)) = view p2 s" ]
    [ "!c p2'.
        view p2(next(do alist)(p2',c)) =
        view p2(next(do(filter alist p1))(p2',c))" ]
```

The second assumption, together with the fact that the first component of h is p1 (the third assumption) will solve the goal. Although this seems a fairly straightforward step, it requires fine manipulation of some of the assumptions and careful rewriting which is performed by the following compound tactic:

```
#e((REWRITE_ASM_TAC [] 4 []) THEN
              ONCE_REWRITE_TAC[PAIR_h] THEN
              PURE_ASM_REWRITE_TAC[] THEN
              REWRITE_TAC[]);;
OK..
goal proved
..... |- view p2(next(nextlist initstate alist)h) =
         view p2(nextlist initstate(filter alist p1))
```

```
Previous subproof:
"view p1 next nextlist initstate alist.h  =
 view p1 next.nextlist initstate filter alist p1  h."
    [ "internally_consistent" ]
    [ "!p1 p2.
        R p1 p2 ==>
        !s t. view p1 next s p1,t'  = view p2 s
        .!p1 s t.
          (view p1 s = view p1 t) ==>
          !t p2. view p1 next s p2,t)
                        = view p1 next t(p2,t)))" ]
    [ "R p1 p2" ]
    [ "view p2 do alist
                  = view p2 do(filter alist p1))" ]
    [ "FST h = p1" = F" ]
```

This completes the proof of the first subgoal of the case analysis step we performed earlier. The second subgoal can be proved along similar lines, thus solving our original goal and proving the desired theorem. As a final step, we add it to the current theory, and bind it to the ML identifier UNWINDING_THM.

```
goal proved
 - !R.
    internally_consistent
     !p1 p2.
      R p1 p2 ==>
       !s t. view p2 next s(p1,t'  = view p2 s
       .!p1 s t.
         view p1 s = view p1 t' ==>
         !t p2. view p1 next s p2,t
        = view p1 next t(p2,t)'  ==> secure R

Previous subproof:
goal proved

#let UNWINDING_THM = save_top_thm ('UNWINDING_THM');;
```

The full details of the proof are given in Appendix A.

5 Proof of Security for the Low Water Mark Model

Given the general theory of security developed in the previous section, it is possible to prove in HOL that a particular system is secure. In this section, we shall describe the proof of security for the Low Water Mark Model using the unwinding theorem.

It is not possible in HOL to use UNWINDING_THM as it stands, because the types command, object, state, output and the functions next and out now need to be given concrete definitions instead of being abstract. A new theory needs to be created where these types and constants (along with other types and constants) are defined. The unwinding theorem, including the auxiliary lemma, needs to be reproved (although this is straightforward, being essentially automatic). This highlights a problem with HOL's treatment of theories. Ideally, we would like to be able to instantiate the original theory by giving some of the abstract types and constants of that theory concrete definitions and automatically inheriting the theorems of the original theory. However, the HOL system does not as yet provide a mechanism for doing this.

5.1 Type and Constant Definitions

We give below the definitions of some of the types which were abstract in the general security theory, and some further types which are needed to describe the Low Water Mark Model. The data objects of the system have the type `object` which is a pair consisting of its contents of type `data` and its classification of type `level`. The states of the system are considered to be mappings from the type `filename`, which is the set of data object identifiers, to the type `object`. The output of a command is considered to be a pair consisting of some data and a boolean value intended to indicate whether the command was successful or not.

```
new_type 0 'level';;
new_type 0 'data';;
new_type 0 'process';;
new_type 0 'filename';;
new_type_abbrev ('object', ":data#level");;
new_type_abbrev ('state', ":filename -> object");;
new_type_abbrev ('output', ":data#bool");;
```

The type `command` is defined using HOL's type definition package, due to Tom Melham. This is an extremely useful facility (with some fundamental limitations which will not concern us here). It is ideally suited to defining new recursive and compound types, because it will automatically provide a number of useful theorems, including an induction theorem. It will also provide a tactic which reduces the proof of a goal with a universally quantified variable of recursive type to a proof of the goal for elements of the base type and a proof of the inductive step for the recursively structured elements of the type. In this case, the type `command` is not recursive, but it is still a good idea to use the type definition package to produce a tactic which splits a goal which is an assertion about commands to three subgoals, one for each type of command — read, write and reset.

```
#let command_Axiom = define_type 'command_Axiom'
    'command = READ filename |
               WRITE data filename |
               RESET filename';;

command_Axiom =
|- !f0 f1 f2.
     ?! fn.
       !f. fn(READ f) = f0 f) /\
       !d f. fn(WRITE d f) = f1 d f) /\
       !f. fn(RESET f) = f2 f)

#let COMMAND_CASES_THM =
     prove_induction_thm command_Axiom;;
COMMAND_CASES_THM =
|- !P.
     !f. P(READ f)) /\ (!d f. P(WRITE d f))
      /\ (!f. P(RESET f)) ==> (!c. P c)

#let COMMAND_CASES_TAC =
   INDUCT_THEN COMMAND_CASES_THM ASSUME_TAC;;
```

The type definition package is also helpful in expressing the view function neatly. As we shall see later, the view of a process is a partial function from filenames to objects, so we need some way of expressing the fact that, for certain filenames, the value of such a function is undefined. Partial functions can be modelled in HOL in the following way. We define a type called `private_state`, which has a distinguished element called UNDEF. Two theorems can be proven about elements of this type – one that the element UNDEF is not equal to an element of the form OBJ `filename` and the other that the constructor OBJ is one-to-one.

```
#let Priv_state_axiom =
       define_type 'Priv_state_axiom'
          'private_state = OBJ object | UNDEF';;

Priv_state_axiom =
 - |- !f e. ?! fn. !p. fn (OBJ p) = f p) /
        fn UNDEF = e)

#let not_obj_undef =
  prove_constructors_distinct Priv_state_axiom;;
not_obj_undef = |- !p. ~(OBJ p = UNDEF)

#let priv_state_one_one =
   prove_constructors_one_one Priv_state_axiom;;
priv_state_one_one =
     |- !p p'. (OBJ p = OBJ p') = (p = p')
```

Some constants also need to be defined: dom denotes the relation on `level` which was called **dominates** in Section 2. The properties of reflexivity, antisymmetry, transitivity and totality for the relation dom can be asserted by including as assumptions to any goals `PO_REFL_DEF`, `PO_ANTISYM_DEF`, `PO_TRANS_DEF`, `PO_TOTAL_DEF`.

```
new_constant ('dom', ":level -> level -> bool");;

PO_REFL_DEF = |- po_refl = !a. dom a a

PO_ANTISYM_DEF = |- po_antisym =
          !a b. dom a b /\ dom b a ==> (a = b)

PO_TRANS_DEF = |- po_trans =
          !a b c. dom a b /\ dom b c ==> dom a c

PO_TOTAL_DEF = |- po_total =
          !a b. dom a b ==> dom b a)
```

The constant `syshigh` denotes a level which is intended to be the level which dominates all other levels. This is asserted by `HIGHMARK_DEF`.

```
new_constant ('syshigh', ":level");;
HIGHMARK_DEF = |- highmark = (!a. dom syshigh a)
```

We also need a function called `process_level` which associates with each process its clearance level. When the output of a command does not include the contents of a file, the constant `null` of type `data` can be used to represent no information.

```
new_constant ('process_level', ":process -> level");;
new_constant ('null', ":data");;
```

We define some auxiliary functions which will be useful in defining the functions `next` and `out`.

```
MK_OBJECT_DEF = |- !a b. mk_object a b = a,b

OBJECT_LEVEL_DEF = |- !X. object_level X = SND X

OBJECT_DATA_DEF = |- !X. object_data X = FST X

UPDATE_DEF = |- !f X s.
      update f X s = (\f'. ((f' = f) => X | s f'))
```

Next and out are supposed to take as their second argument an action, which is a pair consisting of a process and a command. We would like to define next and out using HOL's function definition facility for recursive types by giving the value returned by the functions by cases on the form of the command. However, HOL will only allow such a definition if the variable of type command is a single argument rather than one contained in a pair. To get around this, the functions curry_next and curry_out are defined with the pair of a process and a command split into two arguments and next and out are defined (with an action as a single argument) in terms of curry_next and curry_out. (The purpose of labouring this point is to show that such issues are not always straightforward in HOL).

```
CURRY_NEXT_DEF =
 |- !s p file. curry_next s p(READ file) = s) /\
    !s p d file.
     curry_next s p(WRITE d file) =
      dom(object_level(s file))(process_level p) =>
      update file mk_object d(process_level p))s |
      s) /\
    !s p file.
     curry_next s p(RESET file) =
      dom(object_level(s file))(process_level p) =>
      update file mk_object null_syshigh s |
      s

NEXT_DEF =  |- !s a:action. next s a =
             curry_next s FST a)(SND a)

CURRY_OUT_DEF =
 |- !s p f.
     curry_out s p READ f) =
      dom process_level p)(object_level(s f)) =>
       object_data(s f),T)
       null,F) /\
    !s p d file.
     curry_out s p(WRITE d file) =
      dom object_level(s file))(process_level p) =>
       null,T)    null,F)) /\
    !s p file.
     curry_out s p(RESET file) =
      dom(object_level(s file))(process_level p) =>
       null,T)  null,F))

OUT_DEF = |- !s a:action. out s a =
          curry_out s FST a)(SND a)
```

For the Low Water Mark Model, information is not allowed to flow from p1 to p2 if p2 has a clearance which is strictly lower than p1's clearance. Therefore the security policy can be defined as follows:

```
POL_DEF = |- !p1 p2.
        pol p1 p2 = ~dom(process_level p2)(process_level p1)
```

We also need to give a definition for the view of a process. Rushby gives a construction of a canonical view function for secure systems which at first sight appears a promising candidate for the automation of security proofs. However, there are problems with using such a construction. The definition involves an equivalence relation on action lists, which means that proofs of the conditions of the unwinding theorem would require induction on action lists. This defeats the purpose of the unwinding theorem, because such proofs are long and unwieldy, involving a lot of case analysis. Ideally we need a simple definition of the view of a process which avoids any induction on lists. In many cases this is possible. For the Low Water Mark Model, we can define

the view of a process in a given state as a partial function from filenames to the set of private states. The function is only defined for those filenames whose level is dominated by that of the process and, in that case, the result is the object to which the filename is mapped in the given state lifted into the type of private states. Otherwise the filename is mapped to the element UNDEF.

```
VIEW_DEF = |- !p s. view p s =
  (f. dom.process_level p) object_level s f
        => OBJ(s f) | UNDEF)|
```

5.2 Proof of Security for the Low Water Mark Model

To prove that the Low Water Mark Model is secure, we need to show that it satisfies the three conditions of the unwinding theorem — having first reproved for this model the unwinding theorem in exactly the same way as was done in Section 4.

The first condition (internal consistency of the system) and the second condition of the unwinding theorem are easy to prove by a general strategy of rewriting with definitions, decomposing the goal using STRIP_TAC and performing case analysis until all conditional expressions are removed. The goal is then either proved by rewriting using the assumptions, or proved by finding a contradiction among these assumptions. Some finer manipulation of several of the subgoals and assumptions is required between these steps.

The third condition of the unwinding theorem is much more difficult to prove than the other two, and needs a more detailed discussion. It is this third condition that requires the assertions that dom is a reflexive, antisymmetric, transitive and total order, and that syshigh dominates all other levels. Firstly, the goal is rewritten and stripped until the term obtained is a universal quantification on a variable of type command. We can then use the tactic COMMAND_CASES_TAC, which decomposes this goal into three subgoals corresponding to the three different cases in the type definition of command.

```
g "highmark ... po_refl ... po_antisym ...
        po_trans ... po_total ==>
     !s t u w ... view u s = view u t
          ==> !b.. view u (next s (w,b)) =
                   (view u (next t (w,b)))...)";;


#e (REWRITE_TAC
     [VIEW_DEF;HIGHMARK_DEF;PO_REFL_DEF;
      PO_ANTISYM_DEF; PO_TRANS_DEF;
      PO_TOTAL_DEF; OBJECT_LEVEL_DEF]) THEN
    STRIP_TAC THEN REPEAT GEN_TAC THEN STRIP_TAC
    THEN COMMAND_CASES_TAC);;
OK..
3 subgoals
"!f.
   (\f'.
     dom(process_level u) (SND(next s(w,RESET f)f')) =>
     OBJ(next s w,RESET f f'))
   (\f'.
     dom(process_level u) (SND(next t(w,RESET f)f')) =>
     OBJ next t w,RESET f f')
     UNDEF) "
   [ "!a. dom syshigh a" ]
   [ "!a. dom a a" ]
   [ "!a b. dom a b ... dom b a ==> (a = b)" ]
   [ "!a b c. dom a b ... dom b c ==> dom a c" ]
   [ "!a b. ~dom a b ==> dom b a" ]
   [ "  f.  dom(process_level u)(SND(s f))
          => OBJ(s f)  UNDEF)) =
      (f. (dom(process_level u)(SND(t f))
          => OBJ(t f)  UNDEF))" ]
"!f f.
   (\f'.
     dom(process_level u) (SND(next s(w,WRITE d f)f')) =>
     OBJ(next s(w,WRITE d f)f')
     UNDEF)) =
   (\f'.
     dom(process_level u) (SND(next t(w,WRITE d f)f')) =>
     OBJ(next t(w,WRITE d f)f')
     UNDEF))"
   [ "!a. dom syshigh a" ]
   [ "!a. dom a a" ]
   [ "!a b. dom a b ... dom b a ==> (a = b)" ]
   [ "!a b c. dom a b ... dom b c ==> dom a c" ]
   [ "!a b. ~dom a b ==> dom b a" ]
   [ "(\f. (dom(process_level u)(SND(s f))
                    => OBJ(s f) | UNDEF)) =
     (\f. (dom(process_level u)(SND(t f))
                    => OBJ(t f) | UNDEF))" ]

"!f.
  (\f'.
    (dom(process_level u)(SND(next s(w,READ f)f')) =>
     OBJ(next s(w,READ f)f') |
     UNDEF)) =
  (\f'.
    (dom(process_level u)(SND(next t(w,READ f)f')) =>
     OBJ(next t(w,READ f)f') |
     UNDEF))"
```

```
[ "!a. dom syshigh a" ]
[ "!a. dom a a" ]
[ "!a b. dom a b /\ dom b a ==> a = b" ]
[ "!a b c. dom a b /\ dom b c ==> dom a c" ]
[ "!a b. ~dom a b ==> dom b a" ]
[ "(\f. (dom(process_level u)(SND(s f))
          => OBJ(s f) | UNDEF)) =
   (\f. (dom(process_level u)(SND(t f))
          => OBJ(t f) | UNDEF))" ]
```

These three subgoals all have the same form, so it is likely that the same tactics can be applied to each of them. We shall just look at the second subgoal. Once the outermost quantifiers are stripped, this subgoal requires proving two functions equal. This can be done by showing that the results of the two functions are equal for all values. The theorem `EQ_EXT (|- !f g. (!x. f x = g x) ==> (f = g))` makes such a step valid.

```
#e REPEAT GEN_TAC THEN
    MATCH_MP_TAC EQ_EXT;;;
#OK..
"!x.
   (\f'.
    (dom(process_level u)(SND(next s w,WRITE d f,f'))) ==>
    OBJ(next s w,WRITE d f f')
    UNDEF
  x =
   (\f'.
   (dom(process_level u)(SND(next t w,WRITE d f,f'))) ==>
   OBJ(next t w,WRITE d f)f') |
   UNDEF))
  x"
   [ "!a. dom syshigh a" ]
   [ "!a. dom a a" ]
   [ "!a b. dom a b /\ dom b a ==> (a = b)" ]
   [ "!a b c. dom a b /\ dom b c ==> dom a c" ]
   [ "!a b. ~dom a b ==> dom b a" ]
   [ "((\f. (dom(process_level u)(SND(s f))
                    => OBJ(s f) | UNDEF)) =
      (\f. (dom(process_level u)(SND(t f))
                    => OBJ(t f) | UNDEF))" ]
```

This subgoal can be simplified by once again stripping the outer quantifier and then applying beta conversion. It can then be rewritten using the definition of `next`, producing a fairly complex goal which requires further case analysis in order to be simplified.

```
#e GEN_TAC THEN
      BETA_TAC THEN
       REWRITE_TAC[NEXT_DEF;COPY_NEXT_DEF;
       UPDATE_DEF; MK_OBJECT_DEF] THEN
       CHANGE_ASM_TAC 1
          BETA_RULE o ( th. AP_THM th "x:filename")    ;;
...
" dom
   (process_level u)
   SND
     dom(object_level(s f))(process_level w) =>
      (f'. ((f' = f) => (d,process_level w)    s f'))
     s
    x ) =>
  OBJ
    dom(object_level(s f))(process_level w) =>
     (f'. ( f' = f) => (d,process_level w) ) s f'))
    s
   x)
 UNDEF) =
 dom
  process_level u)
  SND
    dom (object_level(t f) (process_level w) =>
     f'.    f' = f) => (d,process_level w) ) t f')
    t
   x    =>
 OBJ
   dom (object_level(t f    process_level w) =>
     f'.    f' = f  => (d,process_level w) ) t f')
   -
  x
 UNDEF)"
   [ "!a. dom a\a" ]
   [ "!a. dom a a" ]
   [ "!a b. dom a b  \ dom b a ==> (a = b)" ]
   [ "!a b c. dom a b  \ dom b c ==> dom a c" ]
   [ "!a b. ~dom a b ==> dom b a" ]
   [ "~(dom(process_level u)(SND(s x))
                          => OBJ(s x) ( UNDEF) =
     (dom(process_level u)(SND(t x))
                          => OBJ(t x) ( UNDEF)" ]
```

Because the term is so complex, it is difficult to determine by human inspection on what conditions case analysis should be performed. As an automated reasoning tool, it would be useful for HOL to be able to pick out the relevant conditions. HOL does contain a built-in tactic called COND_CASES_TAC which performs case analysis on the outermost conditionals. However, these outer conditionals may contain further conditional expressions which should firstly have been split into cases to put the assumptions in their simplest form. A tactic called COND_DIVE_TAC (written by M. Ozols) is used instead. This tactic searches the term for the first condition of a conditional which does not contain any further conditional expressions unless they involve bound variables occurring within lambda expressions. It performs case analysis on this condition and then attempts to simplify the goal by applying

BETA_TAC and rewriting with the new list of assumptions. If COND_DIVE_TAC is repeatedly applied, all conditional expressions should be eliminated from the goal. When this is done to the above goal, 12 subgoals are produced. Case analysis has in fact been performed on five conditions, which would normally produce 32 subgoals —the tactic automatically solves the other 20 subgoals by rewriting and beta conversion. An example of one of these 12 subgoals is shown below:

```
"UNDEF = OBJ s f "
    [ "!a. dom sysHigh a" ]
    [ "!a. dom a a" ]
    [ "!a b. dom a b /\ dom b a ==> (a = b)" ]
    [ "!a b c. dom a b /\ dom b c ==> dom a c" ]
    [ "!a b. ~dom a b ==> dom b a" ]
    [ "dom(object_level s f)(process_level w)" ]
    [ "x = f" ]
    [ "~dom process_level x (process_level w)" ]
    [ "~dom(object_level t f)(process_level w)" ]
    [ "dom(process_level u)(SND(t f))" ]
    [ "~dom process_level u (SND(s f)) =>
          OBJ s f = UNDEF) = OBJ(t f)" ]
```

It can be seen that the goal can only be proved by finding a contradiction among the assumptions. The overall proof involves solving many subgoals of this form. In the above case, if case analysis is performed on the condition occurring in the first assumption, that assumption will either become `UNDEF = OBJ(t f)` or `OBJ( s f) = OBJ(t f)`. In the first case, the theorem `not_obj_undef` can be used to rewrite the assumption to `FALSE`. In the latter case the theorem `priv_state_one_one` can be used to show `s f = t f`, which, together with the third and the sixth assumptions, gives a contradiction.

```
#e  ASM_CASES_TAC
    "dom process_level u (SND(s:state f))" THEN
    ASM_REWRITE_NTAC 2 [1;3]
         [NOT_OBJ_UNDEF;priv_state_one_one;
            GEN_ALL o NOT_EQ_SYM o
               SPEC_ALL NOT_OBJ_UNDEF]) THEN
    ASM_REWRITE_NTAC 7 [1;4;5] [] THEN
    FALSE_TAC ;;
OK..
goal proved
..........  + UNDEF = OBJ t f

Previous subproof:
11 subgoals
```

The remainder of the proof consists of proving the remaining subgoals in the same way: by rewriting, case analysis, further rewriting, finding contradictions among assumptions etc. The subgoal package will then return us to the point where case analysis was last performed so that the other cases can be solved. Once they are solved, we are returned to the previous case analysis and so on, until all cases have been proved.

The full proof has been completed in HOL. It is rather long and unilluminating, and is given in Appendix A.

5.3 Covert Channels

We conclude this section by considering what happens if we drop the assumption that the relation **dominates** is a total ordering. In this case, it is well-known that the Low Water Mark Model exhibits covert channels, and is not secure. It is interesting to see how to show this in HOL.

Consider two distinct processes p and p' which are completely unrelated to each other (so the ordering is not total). Suppose in the initial state of the system f is a filename whose level dominates both that of p and that of p'. Then both p and p' can successfully write to f. However, if p' writes first, the level of the file becomes that of p' , and p can no longer write to it. Thus process p' has interfered with process p, and so the system is not secure.

The goal is given to HOL as follows:

```
g "~(p' = p)
   ==> ~secure pil";;
```

The proof is straightforward, and is given in Appendix A.

6 Discussion and Conclusions

HOL has reached a certain level of maturity as a research tool, but it is only just beginning to be used in industry. It will be some time before HOL is able to perform program verification, and work in this area is still experimental [6]. However, as we have seen above, HOL can be used with some success to reason about specifications near the top level, and thus to carry out design verification.

Our work has highlighted a number of advantages to working with HOL which make it a useful and versatile tool.

The HOL logic is expressive since it is higher order and polymorphic. As we have seen, it allows theories to be expressed in a natural and succinct way. In many cases this facilitates the understanding of mathematical theories and system specifications.

The soundness and level of mathematical rigour of the HOL system are of particular benefit for reasoning about safety and security critical systems. Once a proof is completed, the user can have a high level of assurance in the outcome.

The built-in inference rules and tactics of HOL are of a fine-grained nature, providing the user with a flexibility not available in many other automated reasoning systems. If a theorem or goal does not exactly fit the form required by a built-in tactic or inference rule, it is possible to carry out quite delicate manipulation until the required form is achieved.

The ability to combine tactics and inference rules using tacticals and the ML language also gives great flexibility and power to the user, who is able to create new tactics designed for use within a particular problem domain, or for goals which have particular characteristics.

The subgoal package allows flexibility in the way theorems are proved. HOL can be used interactively (as a proof assistant) to develop proofs step by step. Proof steps can be 'undone', and the proof state saved at any time, say if we need to prove some side lemma. We can also use it as a proof checker by providing HOL with a complete proof and receiving the proved theorem as output.

On the other hand, carrying out this work has made us acutely aware of some of the shortcomings of HOL.

HOL is a difficult system for a beginner to learn, especially in comparison with other program verification/theorem proving environments such as mEVES [20], Gypsy [21] and MALPAS [22]. As previously noted the first taste of HOL can be quite frustrating, especially without a HOL 'expert' to be a guide. The new user must invest considerable time and effort before simple proofs can be attempted.

The proof presented in the previous section may have given the impression that proofs in HOL are straightforward and quickly constructed — even if they may not always look 'natural'. Unfortunately, this is not the case. A medium-sized proof such as the above is quite laborious and time-consuming to construct, involving a good deal of proof 'exploration'. Often, a vast number of cases must be considered, and one then has the choice of defining a different tactic for each subgoal, or else using a single tactic as a blunt instrument on all the subgoals. Using a single tactic can improve the readability of the proof, but can be quite inefficient — some proofs can take several minutes, or even hours, of computation.

A fundamental difficulty, as we remarked when attempting to go from the general theory of security to the Low Water Mark Model, is that in HOL theories are inherited *en bloc*, and cannot be instantiated to specific instances. Essentially, what is needed is a mechanism for the refinement of types. We plan to examine this question in future work.

We also needed to write a range of fine-grained tactics for rewriting assumptions, and manipulating them in general. While the HOL system allows this to be done without difficulty, it is time-consuming, and we believe that such tactics should already be part of the system.

7 Acknowledgments

We are grateful to Dr Brian Billard for many helpful comments and suggestions. We thank Mr Maris Ozols for providing the functions for manipulating subterms of HOL terms which were used in the proof of security of the Low Water Mark Model, and for useful discussions. We also thank the referees — Mr Damian Marriott, and Dr Peter Lindsay (Software Verification Research Centre, University of Queensland) — for their careful critical reading of the manuscript and many useful comments.

Bibliography

[1] M. J. C. Gordon. A Proof Generating System for Higher Order Logic. Computer Laboratory Technical Report 103, The University of Cambridge, 1987.

[2] M. J. C. Gordon. Why Higher Order Logic is a Good Formalism for Specifying and Verifying Hardware. Computer Laboratory Technical Report 77, The University of Cambridge, 1985.

[3] R. Cardell-Oliver. The Specification and Verification of Sliding Window Protocols. Computer Laboratory Technical Report 183, The University of Cambridge, 1989.

[4] Cambridge Research Centre, SRI International and DSTO Australia. *The HOL System: DESCRIPTION, TUTORIAL and REFERENCE*, 1989.

[5] J. J. Joyce. Using Higher Order Logic to Specify Computer Hardware and Architecture. In D. Edwards, editor, *Design Methodologies for VLSI and Computer Architecture*, pages 129–146. Procs. of the IFIP TC10 Working Conf. on Design Methodology in VLSI and Computer Architecture, Pisa, Italy, September 1988, North-Holland, 1989.

[6] M. J. C. Gordon. Mechanizing Programming Logics in Higher Order Logic. In G. Birtwhistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer-Verlag, 1989.

[7] J Alves-Foss and K Levitt. Verification of Secure Distributed Systems in Higher Order Logic: a Modular Approach using Generic Components. In *IEEE Conference on Security and Privacy (to appear)*, 1991.

[8] A. J. Camilleri. Mechanising CSP Trace Theory in Higher Order Logic. *IEEE Transactions on Software Engineering*, 16:993–1004, 1990.

[9] J. Rushby. Mathematical Foundations of the MLS Tool for Revised Special. Draft internal note, Computer Science Laboratory, SRI International, Menlo Park, Californi, 1984.

[10] B. Billard. Application of a Security Policy to a Low Water Mark Model. RSRE report 86012, RSRE, Malvern UK, 1986.

[11] M.H. Cheheyl, M. Gasser, G.A. Huff, and J.K. Miller. Verifying Security. *Computing Surveys*, 13(3):279–339, 1981.

[12] R. A. Kemmerer. Verification Assessment Study, Final Report Vols 1–5. Technical report, Dept. Computer Science, University of California, Santa Barbara, March 1986.

[13] D. Sutherland. A Model of Information. In *Proceedings 9th National Computer Security Conference*, 1986.

[14] D. McCulloch. A Hookup Theorem for Multilevel Security. *IEEE Transactions on Software Engineering*, 16:563–568, 1990.

[15] J. A. Goguen and J. Meseguer. Unwinding and Inference Control. In *Proceedings 1984 Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society, 1984.

[16] B. Billard. Remarks on Low Water Mark Unwinding. RSRE Memorandum 3991, RSRE, Malvern UK, 1986.

[17] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

[18] A. Wikstrom. *Functional Programming Using Standard ML*. Prentice-Hall International Series in Computer Science, 1987.

[19] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[20] Odyssey Research Associates, Inc. *m-EVES Collected Papers*, September 1989.

[21] W. D. Young. Gypsy Verification Environment User's Manual: Examples of Proof Commands. Technical Report 8, Computational Logic Inc., 1987.

[22] RTP Software Limited. *MALPAS User Guide, Release 4.3*, February 1990.

Appendix A:
Detailed HOL Proofs

We give below, in full, the HOL code for the following:

1. auxiliary tactics needed for the proofs;
2. the proof of the unwinding theorem;
3. the proof of security for the Low Water Mark Model; and
4. an example of a covert channel.

```
%-----------------------------------------------------------------------%
%                  Some Auxiliary Tactics                               %
%-----------------------------------------------------------------------%

let USE_ASM_TAC n infrule =
              ASSUM_LIST (ASSUME_TAC o infrule o (el n));;

letrec ASSUME_LST_TAC thmlst =
        if thmlst = [] then ALL_TAC else
           ASSUME_LST_TAC (tl thmlst) THEN ASSUME_TAC (hd thmlst);;

let CHANGE_ASM_TAC n infrule =
        letrec mk_newasl n infrule l = if l=[] then [] else
            if (n=1) then
                 infrule (hd l) . mk_newasl (n-1) infrule (tl l)
            else (hd l) . mk_newasl (n-1) infrule (tl l)
        in POP_ASSUM_LIST(ASSUME_LST_TAC o (mk_newasl n infrule));;

let ASL_REWRITE_TAC poslst thl =
    ASSUM_LIST
        (\asl. REWRITE_TAC (map (\n. el n asl) poslst) thl);;



let THML_REWRITE_RULE thmlst poslst1 poslst2 otherthms =
    let rewritten = (map (\n. el n thmlst) poslst1) in
    let rewrite_thms = (map (\n. el n thmlst) poslst2) in
      map (\th. if (mem th rewritten) then
       REWRITE_RULE
          ((filter (\x. not th=x) rewrite_thms)@otherthms) th
               else th) thmlst ;;


let ASL_REWRITE_ASL_TAC poslst1 poslst2 thmlst =
   POP_ASSUM_LIST(\asl.
     ASSUME_LST_TAC (THML_REWRITE_RULE asl poslst1 poslst2 thmlst));;

let REWRITE_ASM_TAC thl =
        \(asl,g):goal.
           ASL_REWRITE_ASL_TAC (upto 1 (length asl)) [] thl (asl,g);;

let ASL_REWRITE_ONE_TAC pos poslst thl =
         ASL_REWRITE_ASL_TAC [pos] poslst thl;;

let ASM_MP_TAC n m =  ASSUM_LIST
             (\asl. ASSUME_TAC (MATCH_MP (el n asl) (el m asl)));;
let ASM_LST_MP_TAC n ml = ASSUM_LIST
             (\asl. (ASSUME_TAC (MATCH_MP (el n asl)
                    (CONJL (map (\n. el n asl) ml))))));;
```

```
%----------------------------------------------------------------------%
%                  The Unwinding Theorem                               %
%----------------------------------------------------------------------%


let UNWINDING_THM = prove_thm('UNWINDING_THM',
  "!R. Rpolicy. internally_consistent R ==>
       !u v. R v u ==> !s t. (view v (next s u,t) = view v s
       ! u s t. view u s = view u t) ==>
           !s w. (view u (next s (w,s)) = view u (next t (w,t
    ==> secure R",
  REPEAT STRIP_TAC THEN
  MATCH_MP_TAC UNAFFECTED_VIEW_LEMMA THEN
  ASM_REWRITE_TAC[] THEN
  REPEAT GEN_TAC THEN
  STRIP_TAC THEN
  LIST_INDUCT_TAC THEN
  REWRITE_TAC[FILTER_DEF] THEN      % This proves base case %
  GEN_TAC  THEN                     % Inductive Step %
  COND_CASES_TAC
  THEN
  REWRITE_TAC[DO_DEF;NEXTLIST_DEF] THEN
  CHANGE_ASM_TAC 1 SYM THEN
  ASM_REWRITE_TAC[SYM (SPEC "filter s u" DO_DEF)] THEN
  ASM_REWRITE_TAC[SYM (SPEC "all: action list" DO_DEF)] THEN
  RES_TAC THEN
  REWRITE_ASM_TAC [] THEN
  let apth = SYM (SPEC "h:action"
    INST_TYPE[":process",":*");(":command",":***")] PAIR)) in
    ONCE_REWRITE_TAC[apth]  THEN
  PURE_ASM_REWRITE_TAC[] THEN
  REWRITE_TAC[] ;;


%----------------------------------------------------------------------%
%          Proof of Security for the Low Water Mark Model              %
%----------------------------------------------------------------------%



%               The first condition                                    %


let COND_1 = PROVE
    "po_total ==> internally_consistent",
 REWRITE_TAC[INTERNALLY_CONSISTENT_DEF;VIEW_DEF;PO_TOTAL_DEF] THEN
 STRIP_TAC THEN REPEAT GEN_TAC THEN STRIP_TAC THEN
 COMMAND_CASES_TAC THEN
 REPEAT GEN_TAC THEN
 REWRITE_TAC[OUT_DEF;CURRY_OUT_DEF] THEN
 REPEAT COND_DIVE_TAC THEN
 CHANGE_ASM_TAC 1 (BETA_RULE o (\th. AP_THM th "f:filename"))) THENL
 % 7 SUBGOALS %
 let READ_TAC = (
 ASL_REWRITE_ONE_TAC 1 [2;3] [not_obj_undef;priv_state_one_one;
             ((GEN_ALL o NOT_EQ_SYM o SPEC_ALL) not_obj_undef)]) THEN
(ASM_REWRITE_TAC[]) THEN
(FALSE_TAC) ) and
      WR_RES_TAC1 = (
(ASM_CASES_TAC "dom(process_level u)(object_level((s:state) f))") THEN
```

```
ASM_MP_TAC 6 3;  THEN
ASL_REWRITE_ONE_TAC 3 [1;2] [not_obj_undef;priv_state_the_be;
              GEN_ALL ; NOT_EQ_SYM ; SPEC_ALL  not_obj_undef]  THEN
ASL_REWRITE_ONE_TAC 5 [3;4] []  THEN
FALSE_TAC     and
     WR_RES_TAC1 =
ASM_CASES_TAC "dom:process_level -> object_level bistate  t  "  THEN
ASM_MP_TAC 5 4  THEN
ASL_REWRITE_ONE_TAC 3 [1;2] [not_obj_undef;priv_state_the_be;
              GEN_ALL ; NOT_EQ_SYM ; SPEC_ALL  not_obj_undef]  THEN
ASL_REWRITE_ONE_TAC 5 [3;4] []  THEN
FALSE_TAC     in
     [ READ_TAC; READ_TAC; READ_TAC;
       WR_RES_TAC1; WR_RES_TAC2;
       WR_RES_TAC1; WR_RES_TAC2 ]  ;;;



%                The second condition                                  %


let COND_2 = PROVE (
 "highmark /\ po_refl /\ po_antisym /\ po_trans /\ po_total ==>
    !u s t. (view u s) = (view u t) ==> !c w. (view u (next s (w,c))) = (view u (next t (w,c))))",

 REWRITE_TAC[POL_DEF;HIGHMARK_DEF;PO_REFL_DEF;PO_ANTISYM_DEF;
             PO_TOTAL_DEF;PO_TRANS_DEF]  THEN
 STRIP_TAC THEN REPEAT GEN_TAC THEN STRIP_TAC THEN GEN_TAC  THEN
 COMMAND_CASES_TAC THEN
 REPEAT GEN_TAC  THEN
 REWRITE_TAC[NEXT_DEF;COPY_NEXT_DEF;VIEW_DEF;UPDATE_DEF;
             MK_OBJECT_DEF; OBJECT_LEVEL_DEF;SND;FST]  THEN
 MATCH_MP_TAC EQ_EXT  THEN
 GEN_TAC  THEN
 BETA_TAC  THEN
 REPEAT COND_DIVE_TAC  THENL
 [  ASM_LST_MP_TAC 6 [3;4]  THEN
    ASL_REWRITE_ONE_TAC 1 [2] []  THEN
   FALSE_TAC ;
    ASM_LST_MP_TAC 7 [2;5]  THEN
    ASL_REWRITE_ONE_TAC 1 [3] []  THEN
   FALSE_TAC ;
    USE_ASM_TAC 10  SPEC "process_level v"  THEN
    ASM_LST_MP_TAC 9 [4;1]  THEN
    ASL_REWRITE_ONE_TAC 3 [1;12] []  THEN
   FALSE_TAC ;
    ASM_LST_MP_TAC 7 [2;5]  THEN
    ASL_REWRITE_ONE_TAC 1 [2] []  THEN
   FALSE_TAC ]  ;;;



%                The third condition                                   %


let COND_3 = PROVE (
 "highmark /\ po_refl /\ po_antisym /\ po_trans /\ po_total ==>
    (!u s t.((view u s) = (view u t)
      ==> !c w.((view u (next s (w,c))) = (view u (next t (w,c))))),)",
```

```
REWRITE_TAC[VIEW_DEF;HIGHMAP_DEF;PL_PERL_DEF;PL_ANTISYM_DEF;
            PL_TRANS_DEF; PL_TOTAL_DEF; OBJECT_LEVEL_DEF] THEN
STRIP_TAC THEN REPEAT GEN_TAC THEN STRIP_TAC THEN
DISJAND_CASES_TAC THEN
REPEAT GEN_TAC THEN
MATCH_MP_TAC EQ_EXT THEN
GEN_TAC THEN
BETA_TAC THEN
REWRITE_TAC[NEXT_DEF;CURRY_NEXT_DEF; UPDATE_DEF; MK_OBJECT_DEF;
            OBJECT_LEVEL_DEF] THEN
CHANGE_ASM_TAC 1 (BETA_RULE o th. AP_THM th "x:filename") THEN
ASM_REWRITE_TAC[] THEN
REPEAT (IND_DIVE_TAC) THENL     % produces 12 subgoals %

let tac1 =
 ASM_CASES_TAC "dom(process_level u)(SND((s:state) f))" THEN
 ASL_REWRITE_ONE_TAC 2 [1;3] [not_obj_undef;priv_state_one_one;
            (GEN_ALL o NOT_EQ_SYM o SPEC_ALL) not_obj_undef]) THEN
 ASL_REWRITE_ONE_TAC 7 [1;4;5] [] THEN
 FALSE_TAC    and

    tac2 =
 ASM_CASES_TAC "dom(process_level u)(SND((s:state) f))" THENL
  ASL_REWRITE_ONE_TAC 2 [1;3] [not_obj_undef;priv_state_one_one;
            (GEN_ALL o NOT_EQ_SYM o SPEC_ALL) not_obj_undef] THEN
  ASL_REWRITE_ONE_TAC 7 [1;4] [] THEN
  FALSE_TAC ;
  ASM_MP_TAC 8 5 THEN
  ASM_MP_TAC 9 6 THEN
  ASM_LST_MP_TAC 11 [1;4] THEN
  ASM_LST_MP_TAC 12 [1;4] THEN
  ASL_REWRITE_ONE_TAC 3 [1] [] THEN
  ASM_LST_MP_TAC 14 [1;4] THEN
  ASL_REWRITE_ONE_TAC 8 [1;14] [] THEN
  FALSE_TAC] ) and
    tac3 =
 ASM_CASES_TAC "dom(process_level u)(SND((s:state) f))" THENL
 [ ASL_REWRITE_ONE_TAC 2 [1;3] [not_obj_undef;priv_state_one_one;
          ((GEN_ALL o NOT_EQ_SYM o SPEC_ALL) not_obj_undef)] THEN
  ASL_REWRITE_ONE_TAC 7 [1;4] [] THEN
  FALSE_TAC) ;
  ASM_MP_TAC 8 7) THEN
  ASM_LST_MP_TAC 10 [4:1]) THEN
  ASL_REWRITE_ONE_TAC 3 [1] [] THEN
  FALSE_TAC]    and

          tac4 =
 ASM_CASES_TAC "dom(process_level u)(SND((s:state) f))" THENL
 [ ASL_REWRITE_ONE_TAC 2 [1;3] [not_obj_undef;priv_state_one_one;
          ((GEN_ALL o NOT_EQ_SYM o SPEC_ALL) not_obj_undef)]) THEN
  ASL_REWRITE_ONE_TAC 7 [1;4] []) THEN
  FALSE_TAC) ;
  USE_ASM_TAC 12 (SPEC "process_level u")) THEN
  ASM_LST_MP_TAC 11 [6;1]) THEN
  ASL_REWRITE_ONE_TAC 3 [1;14] []) THEN
  FALSE_TAC] ) and

          tac5 = (
(ASM_CASES_TAC "dom(process_level u)(SND((s:state) f))") THENL
[(ASL_REWRITE_ONE_TAC 2 [1;3] [not_obj_undef;priv_state_one_one;
          ((GEN_ALL o NOT_EQ_SYM o SPEC_ALL) not_obj_undef)]) THEN
```

```
    ASL_REWRITE_ONE_TAC " [5;4] [] THEN
    FALSE_TAC ;
    "ONE_ASM_TAC 11 "SPEC "process_level 1"  THEN
    ASM_LIST_MP_TAC 11 [4;1] THEN
    ASL_REWRITE_ONE_TAC 3 [1;14] [] THEN
   FALSE_TAC] ;; in

  [tac1; tac2; tac1; tac1; tac1; tac3;
   tac1; tac4; tac1; tac1; tac1; tac5]  ;;



%-----------------------------------------------------------------------%
%                          Security Theorem                             %
%-----------------------------------------------------------------------%



let L_W_WATER_MARK_SECURITY = PROVE
    "bitmark    p1_refl    p1_antisym    p1_trans    p1_total ==>
             secure p1",
    STRIP_TAC THEN
    MATCH_MP_TAC "WINDING_THM THEN
    ASSUME_LIST_TAC[CONC_1;CONC_2;CONC_3] THEN
    RES_TAC THEN
    ASM_REWRITE_TAC[] ;;



% ------------------------------------------------------------------- %
%                 Covert Channel                                      %
% ------------------------------------------------------------------- %


% "p" = p1
    p1_refl
   "dom  process_level p1  process_level p" ;
   "dom  process_level p"  process_level p1  \
    dom "object_level  initstate f"  process_level p
    dom "object_level  initstate f"  process_level p"
        ==> "secure p1";;


e  REWRITE_TAC DEFS THEN
   REPEAT STRIP_TAC THEN
   CHANGE_ASM_TAC 1
      SPECL ["p:process" ; "p" ; process"]] THEN
   RES_TAC THEN
   CHANGE_ASM_TAC 1
     SPECL ["[ p" ; process, WRITE a' f);
               p ;  process, WRITE a  f)]";
            "READ f"]] THEN
   ASM_REWRITE_ONE_TAC 1 DEFS THEN

   CHANGE_ASM_TAC 1 BETA_RULE THEN
   ASM_REWRITE_ONE_TAC 1 DEFS THEN
   CHANGE_ASM_TAC 1 BETA_RULE THEN
   ASL_REWRITE_ONE_TAC 1 [5] [PAIR_EQ] THEN
   ASM_REWRITE_TAC []);;
```

DISTRIBUTION

	Copy No.
Defence Science and Technology Organisation	
Chief Defence Scientist)	
Central Office Executive)	1
Counsellor, Defence Science, London	Cnt Sht *
Counsellor, Defence Science, Washington	Cnt Sht *
Scientific Adviser, Defence Central	2
Scientific Adviser to Director Defence Intelligence Organisation	3
Naval Scientific Adviser	4
Air Force Scientific Adviser	5
Scientific Adviser, Army	6
Electronics Research Laboratory	
Director	7
Chief Information Technology Division	8
Chief Communications Division	9
Chief Electronic Warfare Division	10
Chief Guided Weapons Division	11
Special Adviser Combat Systems	12
Research Leader Command and Control	13
Research Leader Intelligence	14
Research Leader Combat Systems	15
PRSC3I	16
Head Command Support Systems Group	17
Head Information Systems Development Group	18
Head Information Processing and Fusion Group	19
Head Software Engineering Group	20
Head Trusted Computer Systems Group	21
Head Architectures Group	22
Head VLSI Group	23
Head Image Information Group	24
Head Combat Systems Integration Group	25
Head Tactical Command Information Systems Group	26
Head Exercise Analysis Group	27
Head Combat Systems Technology Group	28
Head Combat Systems Effectiveness Group	29
A. Cant	30 - 69
K. Eastaughffe	70 - 79
D. Marriott	80
M. Ozols	81
S. Crawley	82
Publications and Component Support Officer	83

Department of Defence

DOCUMENT CONTROL DATA SHEET

Page Classification
UNCLASSIFIED
Privacy Marking/Caveat (of document) N/A

1a. AR Number	1b. Establishment Number	2. Document Date	3. Task Number
AR-006-786	ERL-0577-RR	NOVEMBER 1991	-

4. Title

THE APPLICATION OF HIGHER ORDER LOGIC TO SECURITY MODELS.

5. Security Classification

Document	Title	Abstract
U	U	U

S (Secret) C (Confi) R (Rest) U (Unclass)

* For UNCLASSIFIED docs with a secondary distribution LIMITATION, use (L) in document box.

6. No. of Pages	32
7. No. of Refs.	-

8. Author(s)

DR A. CANT and K. EASTAUGHFFE

9. Downgrading/Delimiting Instructions

N/A

10a. Corporate Author and Address

Electronics Research Laboratory
PO Box 1500
SALISBURY SA 5108

10b. Task Sponsor

-

11. Officer/Position responsible for

Security.....SOERL.....

Downgrading.....

Approval for Release.....DERL.....

12. Secondary Distribution of this Document

APPROVED FOR PUBLIC RELEASE

Any enquiries outside stated limitations should be referred through DSTIC, Defence Information Services, Department of Defence, Anzac Park West, Canberra, ACT 2600.

13a. Deliberate Announcement

NO LIMITATION

13b. Casual Announcement (for citation in other documents)

☑ No Limitation

☐ Ref. by Author , Doc No. and date only.

14. DEFTEST Descriptors

Computer security
Mathematical models
Computer program verification
Low water mark model *
Higher order logic system *

15. DISCAT Subject Codes

1208

16. Abstract

This paper describes the application of the proof assistant HOL (Higher Order Logic) to reasoning about security models. Using Rushby's general framework for security models, we show how the HOL system can prove an unwinding theorem for non-interference of processes at different security levels. The method of unwinding is then applied to the Low Water Mark Model of security. From this analysis, we draw conclusions about the strengths and weaknesses of HOL as a reasoning tool.

Page Classification
UNCLASSIFIED

16. Abstract (CONT.)

17. Imprint

Electronics Research Laboratory
PO Box 1500
SALISBURY SA 5108

18. Document Series and Number	19. Cost Code	20. Type of Report and Period Covered
ERL-0577-RR	711303	ERL RESEARCH REPORT

21. Computer Programs Used

Higher Order Logic

22. Establishment File Reference(s)

N/A

23. Additional information (if required)