Mechanizing Security in HOL

William L. Harrison
Karl N. Levitt
Department of Computer Science
University of California
Davis, CA, 95616

Abstract

Four definitions or "models" of security are described formally and with examples. The formalization into HOL of three of these models is demonstrated. The relative strengths of these definitions is discussed with respect to adequacy, mechanizability, and provability.

1 Introduction

Security has been and continues to be a fertile area for application of verification methods. There are so many subtle ways in which a system can fail to be secure that verification of system security is particularly appropriate. Only the systematic approach offered by verification seems likely, therefore, to inspire confidence about the security of a complicated system.

Many definitions or "models" of security have been proposed in the past. Generally, there are three criteria with which to judge the value of security model. They are :

- Adequacy - does the model admit insecure behavior or rule out secure behavior?
- Mechanizability - how easily can the model be expressed in a formal language?
- Provability - how easily can a system be proven to have the given security property?

The *adequacy* of a security model simply expresses whether the model adequately captures the notion of "secure". There are well known examples of security models which do allow insecure behavior. An example of an *inadequate* model is given in the next section. *Mechanizability* refers to how susceptible the model is to formalization in a language like HOL or EHDM [2]. *Provability* refers to the difficulty of verifying that a particular system has a desired security property. The "non-interference" and "restrictiveness" models described in sections 3 and 4 are concise and elegant, but verifying that a system has either of these properties generally involves lengthy and complicated inductions [1].

2 Access Control

Access control (also known as the Bell-LaPadula security model) is the most familiar of all security models. System objects such as processes and files are assigned a security level[1] (e.g., "unclassified", "secret", or "top secret"), and the following two requirements are enforced:

1. Processes or users may only read from data objects of identical or lower security level.
2. Processes or users may only write to data objects of identical or higher security levels.

The second requirement is necessary so that high-level information can not be accidently or maliciously leaked to a lower level. The access control model of security was inspired by "paper world" security.

One inadequacy of the access control model is that most operating systems require the existence of a "supervisor" which can read and write globally to any data object. Clearly, the existence of a supervisor violates both of the above conditions, so access control implies nothing about the "trustworthiness" of

[1] In general, security levels form a partially ordered set

0-8186-2460-4/92 $03.00 © 1992 IEEE

the supervisor. Another shortcoming of access control is that all information does not flow through *explicit* channels (i.e., files or messages). A high-level "Trojan horse" program could pass information to a low-level process by, for example, through a "covert" channel. A high-level Trojan horse process can signal a low-level process by writing a file to a disk through the supervisor, which is then read by the low-level process.

Another type of covert channel is a "timing channel". For example, if the Trojan horse spots "we attack at dawn" in a high-level file, then it acquires the disk at midnight. If the Trojan horse does not acquire the disk at midnight, it has not found the message in any high-level files. The low-level process can, then, determine whether an attack is imminent by looking at the activity of a certain disk at a certain time.

3 Generalized Non-Interference

A low-level user can deduce information about high-level processing only when the high-level processing has some effect on the low-level user's state. The "effect" can be direct (e.g., reading a secret file) or indirect (e.g., noticing some shared resource like a disk is busy). In this sense, the insecure phenomena derives from the secret user **interfering** with the unclassified processing. In this case, the secret processing is not transparent to the unclassified user. The non-interference security model is an attempt to formalize this notion rigorously.

We define non-interference in terms of a system model called an event system. An **event system** is a quadruple $< E, T, I, O >$ where the set E is all system events, the set T is the set of all valid system traces, and I and O are all system inputs and outputs, respectively.

The standard interpretation of an event system follows. An event can be an internal state transition (e.g., incrementing a register) or a communication event (e.g., a message passed between processors). A valid system trace in T is a sequence of events from E permitted by the system. T defines the behavior of the system. I and O are disjoint subsets of E. It should be noted that an event system is an abstract object completely independent of the standard interpretation. Our event system comes from [1], but there are many similar models in the literature [cf. McCullough and Goguen/Meseguer].

We can now define non-interference formally. Let event system $S = < E, T, I, O >$ and $\mathbf{t1} = \mathbf{a}^\frown \mathbf{d1}$ be a trace in T (NB, $^\frown$ is sequence concatenation). S has the **non–interference property** if for all such **t1** and sequences $\mathbf{a}^\frown \mathbf{d2}$, where **d2** differs from **d1** only in high-level inputs, there is a *trace* $\mathbf{t3} = \mathbf{a}^\frown \mathbf{d3}$, where **d2** and **d3** differ only in high–level outputs. That is, if one changes the high–level inputs in trace **t1** (the result of which is not necessarily a trace), one can find a valid trace **t3** differing from **t1** only in high–level outputs. Or in other words, changes in high–level inputs result only in changes of high–level outputs. S is, in this case, said to have the non-interference property for the low–level view. A definition of non-interference in HOL can be found in the following figure.

```
IS_NON_INTERFERING
  ⊢def ∀ v es. IS_NON_INTERFERING v es =
       (∀ a d1 d2.
          let ev = EVENTS es and
              tr = TRACES es and
              INP = INPUTS es and
              OUT = OUTPUTS es and
              HI_O = OUT DIFF v and
              HI_I = INP DIFF v
          in
          (a SEQ_CONC d1) IN tr ∧
          (SAME_VIEW (ev DIFF HI_I) d1 d2) ⟹
            (∃ d3.
               (a SEQ_CONC d3) IN tr ∧
               (SAME_VIEW (ev DIFF HI_O) d2 d3)))
```

Here, `SEQ_CONC` means sequence concatenation, `HI_I` and `HI_O` are the sets of input and output events, respectively, which are not visible from the view `v`. `SAME_VIEW ev s1 s2` is true if and only if the two sequences of events `s1` and `s2` are identical when restricted to elements of the set `ev`.

4 Restrictiveness

Generalized non–interference provides a satisfactory standard of security for a single system. However, the composition of two non-interfering systems is not necessarily non-interfering. [cf. McCullough]. Composability of a security property is very desirable because of the growing popularity of distributed computing. Hence, lack of composability is a severe drawback of the non-interference security model.

A refinement of non–interference called **restrictiveness** (also known as hook-up security) overcomes this problem. Restrictiveness is a strengthening of non-interference which is preserved under composition. An event system is **restrictive** if, for all traces $\mathbf{a}^\frown \mathbf{b1}^\frown \mathbf{g1}$, where **b1** is a sequence of inputs, and for all sequences $\mathbf{a}^\frown \mathbf{b2}^\frown \mathbf{g1}$ obtained by changing high-level inputs in **b1**, there is a trace $\mathbf{a}^\frown \mathbf{b2}^\frown \mathbf{g2}$ in which **g1** and **g2** differ only in high–level outputs.

At first glance, this definition appears to be little different from non-interference. however, restrictiveness requires that any difference in high level outputs between **g1** and **g2** occur entirely after any changes to the high-level inputs of b1. This additional requirement allows restrictiveness to be composable [5]. An HOL definition of restrictiveness follows.

```
SAME_VIEW
⊢def ∀v t1 t2.
   SAME_VIEW v t1 t2 =
    (t1 SEQ_RESTRICT v = t2 SEQ_RESTRICT v)

IS_RESTRICTIVE_DEF
⊢def ∀v es.
   IS_RESTRICTIVE v es =
    (∀a b1 b2 g1.
      let EV = EVENTS es and
          TR = TRACES es and
          INP = INPUTS es and
          I_D_V = (INPUTS es) DIFF v
      in
       (a IN TR ∧
        SEQ_IN_SET_STAR b1 INP ∧
        SEQ_IN_SET_STAR b2 INP ∧
        SEQ_IN_SET_STAR g1 EV ∧
        ((a SEQ_CONC b1) SEQ_CONC g1) IN TR ∧
        (SAME_VIEW v b1 b2) ∧
        (g1 SEQ_RESTRICT I_D_V = NULL_SEQ) ⟹
         (∃g2.
          (SEQ_IN_SET_STAR g2 EV ∧
((a SEQ_CONC b2) SEQ_CONC g2) IN TR) ∧
          (SAME_VIEW v g2 g1) ∧
          (g2 SEQ_RESTRICT I_D_V) = NULL_SEQ)))
```

5 Information Flow

Information leakage can occur even in a system in which access control is rigidly enforced. An access control policy can prevent direct channels, but can not eliminate so-called "indirect" or "covert" channels.

The following whimsical example of an indirect channel comes from [8]. Imagine a computer system with a high-level user, a low-level user, and a supervisor which enforces access policy. This system has the following instructions:

- READ(i) : X := Register(i)
- WRITE(i): Register(i) := X

where X is a supervisor variable inaccessible by either user. If the high-level user executes READ(1) and no other operation affects X until the low-level user executes a WRITE(1), then the low-level user has potentially gained information about the state of the high-level processings - specifically what Register(1) contained at one point. Note that this leakage was permitted by the access policy because never once did the low-level user attempt to get high-level information directly.

Information flow analysis overcomes the inadequacies of simple access control by detecting *any* kind of information channel – direct or indirect – between users of different security levels.

Information flow is defined in terms of generic instructions like assignment and conditionals. There instructions are sufficiently general to apply to the instruction set of any computer. The predicate FLOW(X,Y) means there is a flow from variable Y to variable X. FLOW(X,Y) is true for each of the following operations:

1. X := f(...,Y,...) (explicit flow)
2. if f(...,Y,...) then X := a (implicit flow)
3. if a, then OP(X,Y,...) , where OP causes flow from Y to X (conditional flow)

Furthermore, FLOW is a transitive relation. That is, if FLOW(X,Y) and FLOW(Y,Z), then FLOW(X,Z).

A system S is secure according to the information flow security model if for every pair of variable X and Y such that FLOW(X,Y), security-level X > security-level Y. There are two properties of this approach that are particularly worth noting. Firstly, information flow analysis is an entirely *syntactic* technique. One can determine the flow patterns between system resources by examining code exclusively. A consequence of the syntactic nature of the analysis is that potential information channels may be identified in which no flow actually occurs. However, data flow analysis will never identify a non-existent channel [8].

In the opinion of the authors, the information flow model of security is equivalent in some sense to generalized non-interference. That is, for some general class of event systems E and $es \in E$, es has the non-interference property between security levels if and only if es does not allow information flow between levels.

6 Summary and Conclusions

For the most part, the four models of security described in this paper have all been formalized in other logics and used to verify systems. The restrictiveness model has been formalized in HOL and used to verify generic components [1].

Ideally, executable code should be verified; but in practice, design specifications and other code abstrac-

tions have been verified. Generally, this happens for two reasons:

- Designs are more susceptible to formalization in logic than code, due to the complexity of programming language semantics.
- Code includes many implementation details which are irrelevant to the intended purpose of the system design.

Currently at UC Davis, we are designing and specifying a distributed operating system kernel with the goal of verifying restrictiveness at the code level.

Acknowledgements

We wish to thank Jim Alves-Foss for the figures and terminology from his paper [1].

References

[1] Jim Alves-Foss and Karl N. Levitt. Verification of Secure Distributed Systems in Higher Order Logic: A Modular Approach Using Generic Components. *1991 IEEE Symposium on Security and Privacy*, 1991.

[2] Computer Science Laboratory, SRI International, Menlo Park, CA. EHDM Specification and Verification System Version 5.0 –Description of the EHDM Specification Language, January 1990.

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

[4] J.A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symposium on Security and Privacy*, pages 75–86, 1984.

[5] D. McCullough. Specifications for multi-level security and a hook-up property. In *Proc. IEEE Symposium on Security and Privacy*, pages 161–166, 1987.

[6] D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symposium on Security and Privacy*, pages 177–186, 1988.

[7] D. McCullough. Foundations of Ulysses: The theory of security. Technical Report RADC-TR-87-222, Odyssey Research Associates, Inc., July 1988.

[8] J. K. Millen. Operating System Security Verification. Technical Report M79-223, The MITRE Corporation, September 1979.