Tad Taylor

VERKSHOP POSITION PAPER

NOTICE: This paper represents the views and opinions of the author and not necessrily those of the DoD Computer Security Center.

INTRODUCTION

While the discipline of formal verification has progressed since the idea was introduced, mastery of the technology has not kept pace. We are in a position of having usable tools near at hand and not having a clear understanding or consensus of what to do with them. This is analogous to having developed the car and no one knows how to drive. We must be fully aware that verification technology is a two-edged sword. It can lead to greatly enhanced products and to assurances that the software is correct or it can shroud the work in mystery, add needless complexity, and instill false confidence. The technology and its practioners are easily capable of 'browbeating' the unwary (including fellow practioners) into believing the correctness of something that formal methods have been applied to.

Because of the current evolutionary state of verification technology, problems show up most often in the areas of formal modeling and design verification. Both areas reside in that semi-rigorous, semi-formal twilight zone where formal meets informal. This is where subtle and potentially fatal flaws may be introduced, and by the very nature of the technology, once they are introduced and accepted at a high level, they must be perpetuated in order to maintain the "correctness" of the work. That is, the very thing that we appeal to for assurance and security is ensuring the propagation of fatal flaws. But before going into detail on these problems, it is necessary to limit the domain of consideration. This paper will only deal with issues and potential solutions for the area of formal models. It would easily require a separate paper to adequately deal with design verification and its related problems. As a matter of fact, the domain will be further restricted to models dealing with security of computer systems. This is where a great deal of the formal modeling work (at least modeling work designed to integrate with formal verification technology) has been concentrated and it is unclear if the techniques and ideas for security models will apply equally well to other areas, such as reliability and functional correctness.

REVIEW OF THE CURRENT SITUATION

In dealing with this issue, the first step is to identify where the problem originates. The reasons are many and varied. One, as mentioned earlier, is that formal models are one of the areas where informal ideas meet the formal world head on. That transition is rarely an easy one. There is a great deal of difficulty in precisely formalizing what one wants. The danger of developing models of "reasonable" appearance that possess dangerous properties is very real. Moreover, there is an inherent danger in reviewing a model for which an english description exists because of the tendency of reviewers to be too easily convinced. Yet, to examine a model in a vacuum takes a great deal of time before it is evident what the model is doing.

Other difficultities arise due to a lack of consensus about the characteristics of a formal model. There is no uniform opinion as to what form a formal model may or should take. One person's solution is another's problem. The issue of just how abstract a model should be is also lacking universal agreement. How much structure should be present, how much detail regarding the application? The degree of flexibility that a model should allow is also open to question. The SRI model is applicable to a greater number of situations than the Bell and LaPadula model is, but is that necessarily a good thing? The approach that a formal security model should take is not an area of uniform agreement either. There's access control and information flow. Is one better than the other? What about the other approaches? Are they better, do they need more consideration?

There is currently no clear winner, as the above discussion indicates. The two most well known models of computer security, the Bell & LaPadula model and the SRI model, are both known to have deficiencies. The Bell & LaPadula model doesn't address all areas of concern, such as covert channels. Furthermore, the model carries detail that is not clearly applicable to a wide variety of situations. A system is restricted to a specific approach when using the Bell & LaPadula model; the architecture must use a reference monitor, it should have entities that are easily identifiable as the subjects and objects, and the constraints of separation should conform to governmental classification levels. It is clearly designed with the intention of being applied to an operating system and to apply it to other situations involves adapting ideas and concepts of the situation to the model. It seems to me that the process should work the other way around, the model and its characteristics should conform to the natural conceptions about the domain.

The SRI model(a.k.a. - Feiertag model, MLS model), on the other hand, doesn't have an adequate formal description. This leaves some rather important features of the model open to mishandling. Some people complain about the lack of a definition as to what constitutes a secure state, while others are content with the notion of secure transitions leaving you in secure states. This model enforces security through control of information flow. But that doesn't always apply to a situation and the approach may break down when applied to actual code or optimizations of verified code. Here again, this model doesn't make sense in all applications.

VIKING APPROACH

Both of these approaches, much less the models, have problems with their application. So the question is "what should be tried next?" One approach that seems promising is that taken by the VIKING project. Here, simplicity is the key, to both the security kernel and the formal model describing the kernel properties. Before saying more about the model, I'll let the abstract from [Hartman 84] provide some background:

> The primary objective of this work is to provide a formally verified implementation of a security kernel targeted specifically for communications applications. For this reason, the VIKING kernel is a "separation kernel"; its requirements are different from previous general purpose operating system security kernels. This kernel will not directly enforce traditional multilevel security (MLS), but will instead form a foundation for the security of application programs. MLS properties enforced in the application will be proved based on the fundamental properties proved in the kernel. The kernel will support concurrency, interprocess communication (IPC), and will enforce isolation between processes. The kernel performs these functions by implementing the Gypsy cobegin and buffer communication mechanisms.

The notion of a separation kernel is from [Rushby 82]. Here, Rushby defines a kernel to mask the fact that virtually separate machines are residing on the same physical base. A pure notion of separability, as Rushby's model is, is a very restrictive policy. It is an easy matter to

modify this to accomodate the explicit and rigid communications paths that the VIKING model will support.

This model, kernel, and appoach conform to what I believe is a very promising approach for the application of formal verification technology. It is simple, the basic model (Rushby's separation model) behind the Gypsy model is highly adaptable to the given situation, and the whole thing integrates well with current verification technology. As one would expect, the model that follows in the Appendix also serves as an example of what I feel is a proper level of abstraction for a model and an acceptable form or approach. But I realize that these later issues are somewhat subjective. I look forward to discussing these and related issues at the Third non-Annual VERKshop.

Tad Taylor DoD Computer Security Center

Hartman, Bret, "A Gypsy-Based Kernel," Proceedings of the 1984 Symposium on Security and Privacy.

Rushby, J.M., "Proof of Separability -- A Verification Technique for a Class of Security Kernels," 5th International Symposium on Programming, turin, Ital, April 82.

APPENDIX VIKING MODEL

```
{                                                          Bret Hartman
                                                           11 Sept 84
                    VIKING Kernel Model

This document describes the formal security model of the kernel.
The model is based heavily on the work of Rushby and his model of "secure
isolation."

This fifth draft makes a few minor changes to correct errors, adds a couple
assertions to aid the proof, and changes m_secure to a stronger
specification.

I/O is described in terms of effects on colors and i/o buffers.
I/O to the machine is also expressed in terms of buffers.
States form a sequence, instead of a set.
Two separate state sequences, one for states after inputs, one for states
after operation.
}

scope viking =

begin

{ Define the elements of the model }

type states = sequence of state;
   type state = pending;

type inputs = buffer of inp;
   type inp = pending;

type outputs = buffer of outp;
   type outp = pending;

type OPERATION = (ENQ, DEQ, OTHER);  { These are operation names, as   }
                                     { opposed to the actual operations}

function NEXTOP(s: state): OPERATION = pending; { total }
   { In machine M, NEXTOP is used to decide which OPERATION will be }
   { invoked next }

function NEXTSTATE(o: operation; s: state): state =
   { Needed since Gypsy can't easily express the use of a function as a
     parameter }
begin
   exit (assume result =
      if NEXTOP(s) = ENQ
         then ENQ_operation(s)
      else if NEXTOP(s) = DEQ
         then DEQ_operation(s)
      else OTHER_operation(s)
      fi fi);
end;

function INPUT_function (i: inp; s: state): state = pending; {total}
        { Input_function describes how input is added to state }

function OUTPUT_function (s: state): outp = pending; {total}
        { Output_function describes how output is derived from state }

function WANT_INPUT (s: state): boolean = pending; {total}
        { True iff machine wants to receive input }


{ Define the machine M = ( S, I, O, OPERATION, NEXTOP,
                           INPUT_function, OUTPUT_function, WANT_INPUT) }

procedure M (var I: inputs<input>; var O: outputs<output>;
             init_state: state) =
begin
   var S_in, S_op: states;   { States after input, States after operation }
   var s0, s1, s2: state;
   var an_input: inp;
   s0:= init_state;
   S_op := S_op <: s0;
   loop

       assert M_secure(S_in,S_op) and          { Security property }

              size(S_in) + 1 = size(S_op) and { Assertions needed in security}
              s0 = S_op[size(S_op)];          { proof                          }

       if WANT_INPUT(s0)
         then receive an_input from I;
              s1:= INPUT_function(an_input, s0);
          else s1:= s0;
       end;
       S_in := S_in <: s1;
       s2:= NEXTSTATE(NEXTOP(s1), s1);
       S_op := S_op <: s2;
       send OUTPUT_function(s2) to O;
       s0 := s2;
   end;
end;

function M_secure (S_in, S_op: states): boolean =
begin
```

```
   exit (assume result iff
      all i: integer,
      all c:color,
      some s0, s1, s2: state,
      some op: operation,

    ( 1 le i and i le size[S_in] and
      s0 = S_op[i] and    { S_op is the sequence of states after operation }
      s1 = S_in[i] and  { S_in is the sequence of states after input }
      s2 = S_op[i+1] and
      op = NEXTOP(s1) )
   ->
    no_state_effects_during_input(s0, s1, c) and
    buffers_depend_only_on_input(s0, s1) and
    state_effects_depend_only_on_private_state(s1, s2, c, op) and
    no_other_state_effects(s1, s2, c) and
    buffer_effects_depend_only_on_buffers(s1, s2, c, op) and
    output_depends_only_on_buffers(s2) and
    private_next_op_depends_only_on_private_state(s1, c) and
    want_input_depends_only_on_buffers(s0) );
end;


{ Abstraction Functions }

type color = pending;

type private_state = pending;

type private_buffer_state = pending;

function COLOUR(s: state): color = {total}  { Which color NEXTOP will be for }
   pending;

function EXTRACTS(s: state; c: color): private_state = {total}
   { Returns c's private view of state }
   pending;

function BUFFERS(s: state): private_buffer_state =
   { Returns view of i/o buffers }
   pending;


{ Queue operations }

function ENQ_operation(s: state): state = pending;
function DEQ_operation(s: state): state = pending;
function OTHER_operation(s: state): state = pending;

{ System Effects }

function INPUT_BUFFER_EFFECTS(b: private_buffer_state; i: inp)
   : private_buffer_state =
{ Describes how buffer changes based upon input }
pending;

function ENQ_DEQ_BUFFER_EFFECTS(p: private_state; b: private_buffer_state)
   : private_buffer_state =
{ Describes how ENQ or DEQ operation affects buffer }
pending;

function OTHER_OPERATION_BUFFER_EFFECTS(b: private_buffer_state)
   : private_buffer_state =
{ Describes how other operations affect buffer }
pending;

function BUFFER_OUTPUT(b: private_buffer_state): outp =
{ Describes how buffer produces output }
pending;

function BUFFER_WANT_INPUT(b: private_buffer_state): boolean =
{ Describes how buffer decides if input is desired }
pending;

function ENQ_DEQ_STATE_EFFECTS(p: private_state; b: private_buffer_state)
   : private_state =
{ Describes how ENQ or DEQ operation affects private state }
pending;

function OTHER_OPERATION_STATE_EFFECTS(p: private_state): private_state =
{ Describes how other operations affect private state }
pending;

function STATE_NEXT_OP(p: private_state): OPERATION =
{ Describes how private state produces next operation }
pending;

{ Security Property Definitions }


function no_state_effects_during_input(s0, s1: state; c: color): boolean =
begin
   exit (assume result iff
      EXTRACTS(s1, c) = EXTRACTS(s0, c));
end;

function buffers_depend_only_on_input(s0, s1: state): boolean =
begin
   exit (assume result iff
      ( BUFFERS(s1) = BUFFERS(s0)
        or
        ( some i: inp,
          s0 = INPUT_function(i, s1)
          and
          BUFFERS(s1) = INPUT_BUFFER_EFFECTS(BUFFERS(s0), i) ) ) );
end;


function state_effects_depend_only_on_private_state(s1, s2: state; c: color;
   op: operation): boolean =
begin
   exit (assume result iff
    ( COLOUR(s1) = c
      ->
      EXTRACTS(s2, c) = if op = ENQ or op = DEQ
                           then ENQ_DEQ_STATE_EFFECTS(EXTRACTS(s1, c),
                                                      BUFFERS(s1))
                           else OTHER_OPERATION_STATE_EFFECTS(EXTRACTS(s1, c))
                        fi ) );
end;


function no_other_state_effects(s1, s2: state; c: color): boolean =
begin
   exit (assume result iff
     ( COLOUR(s1) ne c
       ->
       EXTRACTS(s2, c) = EXTRACTS(s1, c) ) );
```

```
end;

function buffer_effects_depend_only_on_buffers(s1, s2: state; c: color;
   op: operation): boolean =
begin
   exit (assume result iff
    ( COLOUR(s1) = c
      ->
      BUFFERS(s2) = if op = ENQ or op = DEQ
                    then ENQ_DEQ_BUFFER_EFFECTS(EXTRACTS(s1, c),
                                                BUFFERS(s1))
                    else OTHER_OPERATION_BUFFER_EFFECTS(BUFFERS(s1))
                    fi ) );
end;

function output_depends_only_on_buffers(s2: state): boolean =
begin
   exit (assume result iff
       OUTPUT_function(s2) = BUFFER_OUTPUT(BUFFERS(s2)) );
end;

function private_next_op_depends_only_on_private_state(s1: state; c: color)
     : boolean =
begin
   exit (assume result iff
    ( COLOUR(s1) = c
      ->
      NEXTOP(s1) = STATE_NEXT_OP(EXTRACTS(s1, c)) ) );
end;

function want_input_depends_only_on_buffers(s0: state): boolean =
begin
   exit (assume result iff
      WANT_INPUT(s0) = BUFFER_WANT_INPUT(BUFFERS(s0)) );
end;


end;
```

THE RESTRICTED ACCESS PROCESSOR

An Example of Formal Verification

Norman Proctor
SYTEK, Inc.
1225 Charleston Road
Mountain View, California 94043

INTRODUCTION

The formal verification done by SYTEK as part of the development and security assurance of the Restricted Access Processor (RAP) represents important progress toward the verification of medium-scale and large-scale systems in the real world. It is interesting primarily because it was large, rigorous and meaningful and was successfully completed within its original budget. The experience helps to show what can really be done with reasonable resources to verify the security properties of fairly complex systems.

The RAP is being developed by Computer Sciences Corporation (CSC) for NASA. SYTEK did the formal verification for CSC and continues to consult with CSC for the remainder of the verification effort.

The RAP is a fully automated, real-time guard device on a line connecting a system with mixed data to a system with uncleared users. The system with mixed classified and unclassified data is too complex to be trusted to deny the other system access to classified data. The RAP is intended to be simple enough to be verifiable so that it can be trusted to deny access. The line carries message blocks or packets in both directions. The blocks of one message may be interleaved with the blocks of other messages. The RAP must screen each block for each direction.

The RAP verification effort includes a security model specific to the RAP, a formal top-level specification (TLS), a proof that the TLS satisfies the model and an informal demonstration of correspondence between the TLS and the code. CSC designed the RAP and is now developing the code. SYTEK has completed the model, TLS and proof and is advising CSC on the correspondence.

The formal verification proceeded in two full rounds. A model was developed. Then a TLS was developed. Tools for an automated proof were developed, and a proof was attempted. The problems with the proof indicated that changes needed to be made to the model, the TLS and the tools. After all had been revised, the proof was attempted again. During the course of the second proof, only a few more minor adjustments needed to be made to achieve a complete and correct proof. The following narrative