```
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

chronicles the formal verification in more detail.

ROUND ONE

The security policy for the RAP was determined by the government before the RAP development began. The principal points of the policy are that all blocks released by the RAP must be validated and sanitized and that if a block is recognized as invalid, all blocks of its message are logged. Ordinarily, blocks containing classified data or capable of causing classified data to be modified would be recognized as invalid. Validation and sanitization also reduce the bandwidth for covert channels.

The formal security model puts the English-language policy into mathematical language. The basic approach was to consider each input and output to and from the RAP as an event and to develop a predicate for the complete history of all events. The predicate, called the model invariant, is true when the RAP has obeyed the policy throughout its history.

As it was first written, the model used a Pascal-like syntax with universal and existential quantifiers to state the model invariant. Twenty-one predicates and other functions were defined. Most were recursive definitions. They filled about seven pages. Many more pages of prose explained how the model expressed the security policy.

The first model was not a general-purpose model. Because the policy is specific to the RAP, the model reflected the RAP's purpose. The model did manage to stay largely independent of the RAP design. This independence was encouraged by the fact that the model was written while the functional design of the RAP was being developed.

After the functional design was completed, the TLS was written in the specification language SPECIAL to formalize the design specifications in English. SPECIAL was extended to incorporate sequence types and various operators on sequences. The RAP was specified as a state machine with thirteen state variables (hidden VFUN's) and thirty-three transition rules or operations (OFUN's and OVFUN's). The specification filled sixty-seven pages, exclusive of comments.

The basic proof technique was to show by an induction on the initial state and the operations specified in the TLS that the model invariant would always be true for the RAP. Additional invariants would also need to be proved as the model invariant was not strong enough to be proved directly by such an induction. The conjunction of all the invariants to be proved was called the TLS invariant.

A formula generator was developed in LISP. It took as input a parse tree for the TLS and the invariants and produced all the lemmas necessary for the proof. For each invariant, there was a lemma that it was true for the initial state. For each invariant and operation, there was a lemma that the invariant was true after the operation, assuming it was true beforehand.

A theorem prover was also developed in LISP which was intended to find proofs for the lemmas. Given a formula, it would simplify the formula using logically sound transformations. If that did not prove the formula true, it would split it into several cases such that proving all the cases was equivalent to proving the original formula. It would then simplify the cases and output any cases that were still unproved. The user of the theorem prover could then review the unproved cases and decide whether to resubmit them to the theorem prover or to prove them manually. The user could do little to influence what the theorem prover did to try to prove a formula.

For the first proof, the formula generator was used to produce the lemmas for the model invariant only. The theorem prover was able to prove most of the lemmas with unproved cases being recycled up to three times. Unfortunately, the lemmas proved were only the easy ones. Many hundreds of cases were proved for the unproved lemmas, but hundreds more cases were left unproved. Because the theorem prover had subjected them to many transformations, the unproved cases appeared largely unrelated to their original lemmas and were extremely difficult to analyze.

The theorem prover's work on the unproved lemmas was ignored. The proofs of those lemmas were sketched manually. In developing the sketches, the need for a specific new invariant would occasionally become apparent. A few more invariants were added in order to prove some of the new invariants. The manual process eventually identified an adequate and provable TLS invariant.

The difficulties with the theorem prover made it clear that several changes would have to be made in order to reduce drastically the manual portion of the proof. First, the theorem prover would have to allow considerable guidance by the user. The existential quantifiers in the model would have to be eliminated wherever possible. Finally, any expressions within the TLS that the proof would deal with in detail were worth simplifying as much as possible.

ROUND TWO

For the second proof, a new LISP theorem prover was written. It might be more accurately called a theorem checker since it would only transform a formula a little bit at a time and required a rather explicit command from the user each time. The new theorem prover was intended to allow the user to control the proof so

closely that the results of the user's commands would usually be easy to predict and always be easy to understand. Commands were provided that allowed the theorem prover to analyze those constructs in the SPECIAL language that were used most extensively in the TLS.

The model was revised. The new model was written using SPECIAL syntax. Some definitions were rephrased so that the theorem prover could more easily handle them, but the biggest change was that all existential quantifiers were eliminated. This was done essentially by specifying exactly where the instance could be found that would satisfy the quantified predicate. This unfortunately made the model much more dependent upon details of the RAP design. The risk is that some of those details will change before the RAP is completed or even after it is first completed.

The new model was considerably bulkier than the first one. It had thirty-one predicates and functions instead of twenty-one and took about eleven pages instead of seven.

The TLS was also revised. The division of its functionality into operations was revised so that those operations could be specified with simpler expressions. A few changes were also made because of design changes since the first TLS was written. The new TLS although simpler from the theorem prover's point of view was larger from a reader's point of view. It had fifteen state variables, thirty-seven operations and eighty-nine pages altogether.

For the second proof, an augmented TLS was created by combining the model and the TLS and adding fourteen more invariants for the full TLS invariant. The parse tree of the augmented TLS was input to the formula generator. The formula generator had been changed slightly to eliminate trivial lemmas. It produced 303 lemmas from the fifteen invariants and thirty-seven operations.

Sketches were developed manually to help the user prove the lemmas with the theorem prover. For all but a dozen lemmas, the sketch simply identified the information available for the proof which was also useful for the proof. With those sketches a user only needed to be familiar with the prover to find a proof. The user did not need to understand what the invariant or the operation meant. These proofs typically took about ten or fifteen commands each. A few took up to two hundred commands but were still fairly straightforward.

The dozen exceptional lemmas were not straightforward. They required much more interaction between the theorem prover's user and the specification writer who could understand what was and was not true and why. Four of these proofs took over two thousand commands each.

Occasionally, while proving a lemma, especially a difficult one, a formula would appear that seemed to be unprovable. Sometimes this was because too much information had been discarded in the course of the proof. Other times, however, what had actually been found was an error in the augmented TLS. The error was often in the phrasing of the changes made to the model to eliminate the existential quantifiers. Errors were also found in the fourteen invariants added to make the TLS invariant. The errors were mostly subtle ones best seen under the glaring light of a failure in the proof. Fixing an error found proving one lemma usually required a little reworking of the already completed proofs of other lemmas.

When all the lemmas had been proved, the proofs were all run again to generate a report of the proofs. In this background mode, the proofs of the 303 lemmas took about twenty hours of CPU time on a VAX 750. In the interactive mode used to find the proofs, the response time between commands was typically less than ten seconds and was only occasionally annoying.

The theorem prover has a special command that allows any predicate to be asserted as axiomatically true. The truth must then be argued with a manual proof. The proofs of the 303 lemmas used axioms 141 times, but there were only twenty-four different axioms. For all but one of the axioms, no proof was necessary because the axiom's truth was obvious. A typical axiom was that "x = (x + 1) - 1". This axiom was needed because the theorem prover had no rules for simplifying arithmetic expressions. Such simplifications were needed only a few times so axioms were used instead of adding power to the theorem prover. The exceptional axiom required a three-page manual proof.

CONCLUSION

Besides the axioms, every command was necessarily a logically sound transformation. Since the axioms used were in fact all true, the proofs developed using the theorem prover's commands are all sound. They are also exceptionally detailed proofs. Since proofs were found for all the lemmas, the proof that the TLS invariant is truly an invariant of the TLS is extremely rigorous. Finally, since the TLS invariant includes the model invariant, that rigorous proof shows that the RAP as specified by the TLS satisfies its security policy as expressed in the model. The formal verification produced a solid result for an unusual policy and a moderately complex design.

The development of both versions of the model took roughly one staff year. So did the two versions of the TLS. The development of the tools and the proofs combined took roughly one and a half staff years. Thus, the formal part of the RAP verification cost "only" three and a half staff years. The cost in machine time and disk space is not significant in comparison to the labor cost. Although only a manual proof had been envisaged in the original budget, the verification including the development of the tools was completed within that original budget. The automated proof is considerably more valuable for security assurance than a completely manual proof could have been.