SRI International

CSL Technical Note • March 2004

A Separation Kernel Formal Security Policy in PVS

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

This research was supported by NASA Langley Research Center under contract NAS1-00079.

Computer Science Laboratory•333 Ravenswood Ave.•Menlo Park, CA 94025•(650) 326-6200 •Facsimile: (650) 859-2844

Abstract

Greve, Wilding, and Vanfleet [GWV03] present an ACL2 formalization of a security policy for a separation kernel, and validate its utility by using it to support the verification of a simple application. This note reworks their development in PVS and uses the exercise to offer some comparisonns between PVS and ACL2.

Contents

Contents ii

1 Introduction 1

2 The Security Policy 1

2.1 Consistency 6

3 A Firewall Application 8

4 Conclusion 17

References 19

1 Introduction

A *separation kernel* [Rus81] is a minimal operating system kernel that does nothing more than is necessary to create separate execution environments (usually called *partitions*) and controlled communication channels between them. A separation kernel typically manages hardware protection mechanisms, fields interrupts, and performs scheduling; the full functionality and API of an operating system is built above this and typically is provided separately within each partition. Applications are constructed within and across partitions. This approach structures the assurance arguments for application security into two parts: one concerned with the separation and controlled communications created by the separation kernel, and the other concerned with how these resources are used by a given application. The advantage claimed for this approach over more monolithic variants is that the minimality of the separation kernel enables construction of a credible assurance argument and, in particular, renders formal arguments economically and intellectually feasible.

In this approach, the crucial interface in the overall security assurance argument is the security policy of the separation kernel: the separation kernel must be verified to enforce this policy, while verification of application security will use the policy as its bedrock. Greve, Wilding, and Vanfleet [GWV03] present a security policy for a separation kernel and argue that it reflects intuitive requirements, is at least as strong as policies employed in previous efforts, and is adequate to support at least one representative application. They formalize these demonstrations using the ACL2 [KMM00] theorem prover. In this note, we replicate their development using the PVS theorem prover [ORSvH95, OSRSC98]..

PVS and ACL2 are comparably powerful as theorem provers, but they support very different styles of specification and proof. Whereas ACL2's specification logic is unquantified and untyped (it is the applicative subset of Common Lisp), PVS's is not only quantified but higher-order (i.e., it allows quantification over functions and predicates, not just individuals) and it is not merely typed, but has full dependent typing with predicate subtypes. The applicative fragment of PVS can be executed, just like ACL2, but its execution engine performs full static analysis and is able to compile destructive updates whenever these are safe, resulting in much higher performance than ACL2.[1] And whereas the user guides ACL2's prover only implicitly through the selection and ordering of lemmas to be proved, PVS's theorem prover is fully interactive. The separation kernel security policy and the proof that it can support a "firewall" application are an attractively interesting but simple example on which to illustrate these differences.

2 The Security Policy

The abstract security policy is specified for a state machine comprised of *partitions*, *segments*, and so on, and a relation on segments called *dia* (for "direct interaction allowed").

[1] ACL2 also can perform destructive updates, but these require special annotations to be added by hand.

The security policy is specified as an axiom constraining the behavior of this state machine. Axioms should always be used with caution, since they can introduce inconsistencies. One way to establish that an axiomatically defined theory is consistent is to exhibit a model for the theory.

In the ACL2 specification, this is all accomplished by an `encapsulate` construction. Because ACL2 is untyped, stating that `current` is a function that takes a `state` and returns a `partition` requires the following elements in the encapsulation.

```
((allparts) => *)

((current *) => *)

(defthm current-is-partition
  (member (current st) (allparts)))
```

In PVS, on the other hand, we simply state that `states` and `partitions` are nonempty types, that `s` is a variable of type `states`, and then introduce current as an uninterpreted function with the desired signature.

```
states, partitions: TYPE+

s: VAR states

current(s): partitions
```

Building on this, the full collection of types and signatures needed to specify the kernel are shown below.

```
kernel: THEORY
BEGIN

  states, values, partitions, segments: TYPE+

  s, t: VAR states
  a, b: VAR segments
  A: VAR set[segments]
  x: VAR values
  p: VAR partitions

  current(s): partitions
  segs(p): set[segments]
  select(s, a): values
  dia(a): set[segments]
  next(s): states
```

The intended intuitive interpretations are that exactly one partition is `current` in any state, and that the machine proceeds from state to state by executing its `next` function.

Each partition `p` is associated with a set of memory `segments` given by `segs(p)`. Note that `segs(p)` and `segs(q)` need not be disjoint. The construction `set[segments]` is actually a shorthand for `sets[segments].set`, where the square brackets indicate that `segments` is a parameter to the theory `sets` (from the *prelude* library that is built-in to PVS) and `set` is the type of sets over this parameter. States assign `values` to segments; the value of segment `a` in state `s` is given by `select(s, a)`. For any segment `a`, there is a set of segments `dia(a)` whose values in one state are allowed to influence the value of `a` in the `next` state.

The state machine executes by repeatedly applying its `next` function as specified in the definition of the `run` function. Because this is a recursive function, we supply a `MEASURE` that PVS will use to establish termination.

```
  n: VAR nat

  run(s, n): RECURSIVE states =
   IF n=0 THEN s ELSE run(next(s), n-1) ENDIF
  MEASURE n
```

PVS generates proof obligations called TCCs ("type correctness conditions") to ensure conditions such as termination and type-correctness when these cannot be determined by traditional static analysis. The definition of `run` generates the following TCCs that are discharged automatically by the PVS theorem prover.

```
% Subtype TCC generated (at line 22, column 64) for  n - 1
    % expected type  nat
  % proved - complete
run_TCC1: OBLIGATION
  FORALL (n: nat): NOT n = 0 IMPLIES n - 1 >= 0;

% Termination TCC generated (at line 22, column 51)
% for run(next(s), n - 1)
  % proved - complete
run_TCC2: OBLIGATION
  FORALL (n: nat): NOT n = 0 IMPLIES n - 1 < n;
```

To specify separation in way similar to that used in ACL2, we first need a predicate `eqvals` that tells whether two states have the same values for all segments in a given set `A`. We use the higher-order and predicate subtyping features of PVS to specify this in a powerful way. Note that `=>` is a synonym for the keyword `IMPLIES`.

```
  eqvals(A): (equivalence?[states]) = LAMBDA s,t:
    FORALL a: member(a, A) => select(s, a) = select(t, a)
```

This definition states that `eqvals(A)` is an equivalence relation on states: `equivalence?` is a predicate defined in the PVS prelude and a predicate enclosed in

parentheses is a type. The benefits of this style are succinct specifications (because much of the specification is embedded in the types and therefore need not be repeated at each use) and increased proof automation (because the PVS prover can easily access the predicate associated with the type). Observe that the value of `eqvals(A)` is a relation (that is, a function from pairs to booleans), so it is a *higher-order* function; to check that states `s` and `t` have the same values for all segments in `A`, we write `eqvals(A)(s, t)`. We could just as well have defined a simple predicate `eqvals(A, s, t)`, but the advantage of the *Curried* form is that the component `eqvals(A)` is independently useful.

It is not obvious that the body of `eqvals(A)` defines an equivalence relation, so PVS generates the following TCC requiring us to prove this fact. The proof is discharge automatically by the default strategy.

```
% Subtype TCC generated (at line 26, column 36) for
    % LAMBDA s, t: FORALL a: member(a, A) =>
       %  select(s, a) = select(t, a)
    % expected type  (equivalence?[states])
  % proved - complete
eqvals_TCC1: OBLIGATION
  FORALL (A: set[segments]):
    equivalence?[states]
        (LAMBDA s, t:
           FORALL a: member[segments](a, A) =>
                       select(s, a) = select(t, a));
```

Now `separation` is specified by the following axiom.

```
separation: AXIOM
  LET segs = intersection (dia(a), segs(current(s))) IN
   current(s) = current(t)
   AND select(s, a) = select(t, a)
   AND eqvals(segs)(s, t)
 =>
   select(next(s), a) = select(next(t), a)
```

This axiom says that if the same partition is current in states `s` and `t`, segment `a` has the same value both states, and all the segments that are both in the current partition and allowed directly to influence `a` are the same in both states, then the value of `a` must be the same in the successors of the two states. Another way to interpret this is that the only segments that can influence the `next` value of `a` are those that are allowed to do so and that also are in the current partition.

In the ACL2 treatment, separation is specified as follows, where `selectlist` is a recursively-defined function that creates the list of values associated with a given list of segments in a given state.

```
(defthm separation
  (let ((segs (intersection-equal (dia seg)
                                  (segs (current st1)))))
    (implies
     (and
      (equal (selectlist segs st1) (selectlist segs st2))
      (equal (current st1) (current st2))
      (equal (select seg st1) (select seg st2)))
     (equal
      (select seg (next st1))
      (select seg (next st2))))))
```

The main difference to note between this and the PVS specification is that PVS supports what might be considered a more mathematical statement, using sets and quantification, whereas ACL2 uses a more programming-like style, involving the construction and comparison of explicit lists.

Greve, Wilding, and Vanfleet provide additional justification for the choice of policy expressed in `separation` by showing that it implies three variants that have been used in other developments. The first of these concerns the special case when there are no segments that are both in the current partition and allowed directly to influence `a`.

```
exfiltration: LEMMA
  current(s) = current(t)
  AND select(s, a) = select(t, a)
  AND empty?(intersection(dia(a), segs(current(s))))
=>
  select(next(s), a) = select(next(t),a)
```

This lemma is proved automatically by PVS using the following proof strategy.

```
(grind-with-lemmas :lemmas "separation")
```

Here, `grind-with-lemmas` is the general-purpose strategy for automated proofs using additional named formulas (which are usually lemmas, hence the name), with the `separation` axiom named as the only additional formula to be used. The proof language of PVS is Lisp-like, and strategy invocations such as that above can be typed directly into its command buffer and the results observed in the same buffer. Proof scripts are saved and can be rerun, either automatically, or in a "single-step" interactive mode.

Another variant formulation of separation considers the case where all segments in the current partition have the same values in both states.

```
mediation: LEMMA
  current(s) = current(t)
  AND select(s, a) = select(t, a)
  AND eqvals(segs(current(s)))(s, t)
=>
  select(next(s), a) = select(next(t), a)
```

This lemma is proved in the same way as the previous one.

The third variant is the same as the previous one, except that segment `a` is a member of the current partition.

```
infiltration: LEMMA
  current(s) = current(t)
  AND member(a, segs(current(s)))
  AND eqvals(segs(current(s)))(s, t)
=>
  select(next(s), a) = select(next(t), a)

END kernel
```

This lemma is proved in the same way as the other two variants.

2.1 Consistency

It is, of course, possible that the reason these lemmas are proved so easily is that our `kernel` theory harbors an inconsistency. Provided all TCCs are discharged, PVS guarantees that the only way to introduce an inconsistency is with an axiom (i.e., all definitional constructions in PVS yield conservative extensions). To establish that an axiomatic theory is consistent, we must demonstrate that it has a model. This is accomplished in PVS using the mechanism of theory interpretations [OS01, Sho67]. A theory interpretation translates the types and uninterpreted constants of one ("upper") theory into types and values from another ("lower") theory. For the lower theory to truly *interpret* the upper theory, the translations of the axioms of the upper theory must be provable in the lower theory. If the lower theory is purely constructive (i.e., has no axioms), this demonstration establishes consistency of the upper theory.

An interpretation of the `kernel` theory onto a constructive lower theory is specified in PVS as follows. The translations of the types and uninterpreted constants are specified as "assignments" between the `{{...}}` braces of the `IMPORTING` clause; we give the interpretation a name (`kinst`) so that we can refer to it later.

```
kernel_consistency: THEORY
BEGIN
  addrs, vals: TYPE = upto(10)
  mstates: TYPE = [# pc: addrs, mem: [addrs -> vals] #]

IMPORTING kernel {{
  states     := mstates,
  values     := vals,
  partitions := addrs,
  segments   := addrs,
  current    := LAMBDA (s:mstates): s`pc,
  segs       := LAMBDA (p:addrs): emptyset[addrs],
  select     := LAMBDA (s:mstates, a:addrs): s`mem(a),
  dia        := LAMBDA (a:addrs): singleton(a),
  next       := id[mstates]
}} AS kinst

END kernel_consistency
```

Here, our lower theory is a "machine" whose states consist of a "program counter" `pc`, and a "memory" `mem` which is a mapping from `addrs` to `vals`; these types are synonyms for the small integer range `upto(10)` (i.e., `0,...,10`). The `[#...#]` brackets indicate a record type in PVS; here the field identifiers are `pc` and `mem` and dereferencing the `pc` field of an `mstates` record `s` is denoted `s`pc`. The `[...->...]` construction indicates a function type.

The assignments in the `{{...}}` braces specify the translations of the types of the `kernel` theory into the types of this "machine," and translations of its uninterpreted constants into expressions over those types. Since we are interested only in establishing consistency, there is no need for our translation to be "realistic," so we map the `next` state function to the identity function on machine states `id[mstates]`, we assign the empty set as the value of `segs(p)`, and a singleton set as the value of `dia(a).` (Other simple assignments would serve equally well; these were chosen to parallel those used in the ACL2 version.)

Typechecking the `kernel_consistency` theory generates warning messages and/or TCCs (i.e., proof obligations). The warnings are present when the list of assignments is incomplete and a full theory interpretation cannot be generated. Once the warnings are no longer present, the typechecker generates TCCs that must be discharged to ensure the soundness of the interpretation. These TCCs are the translated forms of the axioms present in the upper (i.e., `kernel`) theory. Since the only axiom is `separation`, the following is the only TCC.

```
% Mapped-axiom TCC generated (at line 163, column 10) for
    % kernel
    %        states := mstates,
    %          values := vals,
    %          partitions := addrs,
    %          segments := addrs,
    %          current := LAMBDA (s: mstates): s‘pc,
    %          segs := LAMBDA (p: addrs): emptyset[addrs],
    %          select := LAMBDA (s: mstates, a: addrs): s‘mem(a),
    %          dia := LAMBDA (a: addrs): singleton(a),
    %          next := id[mstates]
  % proved - complete
IMP_kernel_separation_TCC1: OBLIGATION
  FORALL (a: addrs, s, t: mstates):
    (s‘pc = t‘pc AND
      s‘mem(a) = t‘mem(a) AND
       (FORALL (b: addrs):
          member[addrs]
              (b, intersection[addrs](singleton[addrs](a),
                                      emptyset[addrs]))
           => s‘mem(b) = t‘mem(b)))
     => id[mstates](s)‘mem(a) = id[mstates](t)‘mem(a);
```

This TCC is discharged automatically by the default proof strategy. As the `kernel_consistency` theory is purely constructive, the validity of this TCC ensures consistency of the `kernel` theory.

3 A Firewall Application

To demonstrate utility and realism of their separation policy, Greve, Wilding, and Vanfleet show that it can be used to support a simple application that they call a “firewall.” Data containing no sensitive material can be described as “black” and the firewall must ensure that all the data belonging to a distinguished partition `B` is always black. To ensure this the `dia` function is set up in a specific way. In particular, there is another distinguished partition `F` and a distinguished segment called `outbox`, and `dia` is such that “if there is a segment in partition `B` that is writable from a segment that is in a non-`B` partition, then it has the name `outbox` and it is only writable from segments that are both in partition `F` and not in partition `B`” [GWV03, page 10] (i.e., all information flow from non-`B` to `B` is through `outbox` and comes from `F`). In addition, “when partition `F` is executing, memory segment `outbox` does not transition from black to non-black.” We can specify these aspects of the design as follows.

```
firewall: THEORY
BEGIN

  sk: THEORY = kernel

  s, t: VAR states
  p: VAR partitions
  a, b: VAR segments
  A: VAR set[segments]
  x: VAR values

  F, B: partitions
  outbox: segments

  black(s, a): bool

  dia_setup: AXIOM
    member(a, dia(b))
    AND member(b, segs(B))
    AND member(a, segs(p))
    AND p /= B
  =>
    b = outbox
    AND p = F
    AND NOT member(a, segs(B))

  firewall_blackens: AXIOM
    current(s) = F AND black(s, outbox)
  =>
    black(next(s), outbox)
```

The theorem we wish to prove is the following: if all segments of B are black, then they stay that way.

```
  blackset(s, A): bool = FORALL a: member(a, A) => black(s, a)

  n: VAR nat

  firewall_works: THEOREM
    blackset(s, segs(B))
  =>
    blackset(run(s, n), segs(B))

END firewall
```

To prove this theorem, Greve, Wilding, and Vanfleet suppose that the firewall is equipped with a function `blacken`[2] that renders all the segments in the current state black. We further suppose that the `next` states of all-black states are all-black also. Then they use the separation axiom to show that if all the segments of `B` are black, then the next state is indistinguishable to `B` from the next state of the blackened version of the original state. The `blacken` function need not be implemented: it is an artifact of proof. However, we do need to axiomatize its properties and to ensure they are consistent.

We begin by introducing `blacken` as an uninterpreted function on states, then define `allblack(s)` as a shorthand for a state whose segments are all black. Then we state the crucial property of `blacken` as the axiom `blacken_blackens.` The assumption that all-black states stay that way is formalized in the axion `spontaneous_generation`.

```
blacken(s): states

allblack(s): bool = blackset(s, fullset[segments])

blacken_blackens: AXIOM allblack(blacken(s))

spontaneous_generation: AXIOM allblack(s) => allblack(next(s))
```

Next, we need some additional properties of `blacken`; these are formalized in the following three axioms.

```
blacken_select: AXIOM
  black(s, a) => select(s, a) = select(blacken(s), a)

blacken_current: AXIOM current(s) = current(blacken(s))

black_funof_seg: AXIOM
  select(s, a) = select(t, a) => black(s, a) = black(t, a)
```

The first of these says that `blacken` does not change the value of an already black segment, while the second says that the identity of the current partition is unaffected by `blacken.` The third axiom says (indirectly) that the value of a segment determines whether it is black or not.

Equipped with these axioms, the proof of `firewall_works` is straightforward. We begin by instructing the prover to use induction on the parameter `n`, and then invoke the `separation` axiom.

```
(induct-and-simplify "n" :exclude "blackset")
(hide 2)
(expand "blackset" +)
(skosimp)
(lemma "separation")
(beta *)
```

[2] Actually, they call this function `scrub`.

This series of proof commands yields the following *sequent*.

```
{-1}  FORALL (a: segments, s, t: states):
        (current(s) = current(t) AND
          select(s, a) = select(t, a) AND
           (FORALL b:
              member(b, intersection(dia(a), segs(current(s)))) =>
               select(s, b) = select(t, b)))
         => select(next(s), a) = select(next(t), a)
[-2]  member(a!1, segs(B))
[-3]  blackset(s!1, segs(B))
  |-------
[1]   black(next(s!1), a!1)
```

Sequents are the form in which PVS displays its proof state to the user. A sequent consists of zero or more negatively numbered *antecedents* (or *premises*) above the `|--------` line, and zero or more positively numbered *consequents* (or *conclusions*) below. The numbers are used to refer to their formulas; the `{.}` brackets (as opposed to `[.]`) around a number simply indicate that this formula was affected by the previous proof command. Identifiers with `!` in them such as `a!1` indicate *Skolem constants*. The interpretation is that the conjunction of the antecedents must be shown to imply the disjunction of the consequents; proof commands are used to transform (and possibly split) the sequent until this fact is obvious to the prover. Sequents form a tree (which can be displayed graphically); one of the unproved leaves of this tree is selected as the *current* sequent and is presented to the user; proof commands apply to the current sequent.

We instantiate the axiom as suggested in the earlier proof sketch, and then do a case-split, which yields four cases.

```
 (inst - "a!1" "s!1" "blacken(s!1)")
 (ground)
```

The first case is the following.

```
Applying propositional simplification and decision procedures,
this yields  4 subgoals:
firewall_works.1 :

{-1}  select(next(s!1), a!1) = select(next(blacken(s!1)), a!1)
{-2}  segs(B)(a!1)
[-3]  blackset(s!1, segs(B))
  |-------
[1]   black(next(s!1), a!1)
```

This case is easily discharged by first invoking the appropriate instantiation of the `spontaneous_generation` axiom, and then leaving PVS to deal with the rest of the proof automatically, using the two axioms `black_funof_seg` and `blacken_blackens`.

```
   (lemma "spontaneous_generation" ("s" "blacken(s!1)"))
   (grind-with-lemmas
          :lemmas ("black_funof_seg" "blacken_blackens")))
```

We are then presented with the second case.

```
Grinding away with the supplied lemmas,,

This completes the proof of firewall_works.1.

firewall_works.2 :

{-1}  segs(B)(a!1)
[-2]  blackset(s!1, segs(B))
  |-------
{1}   current(s!1) = current(blacken(s!1))
[2]   black(next(s!1), a!1)
```

This sequent is discharged by the `blacken_current` axiom.

```
  (rewrite "blacken_current")
```

We are then presented with the third case.

```
Found matching substitution:
s: states gets s!1,
Rewriting using blacken_current, matching in *,

This completes the proof of firewall_works.2.

firewall_works.3 :

{-1}  segs(B)(a!1)
[-2]  blackset(s!1, segs(B))
  |-------
{1}   select(s!1, a!1) = select(blacken(s!1), a!1)
[2]   black(next(s!1), a!1)
```

This sequent is an obvious consequence of the axiom `blacken_select`.

```
  (grind-with-lemmas :lemmas ("blacken_select"))
```

We are now presented with the final case.

```
Grinding away with the supplied lemmas,,

This completes the proof of firewall_works.3.

firewall_works.4 :

{-1}  segs(B)(a!1)
[-2]  blackset(s!1, segs(B))
  |-------
{1}   FORALL b:
        intersection(dia(a!1), segs(current(s!1)))(b) =>
         select(s!1, b) = select(blacken(s!1), b)
[2]   black(next(s!1), a!1)
```

This is the only case that depends on us setting up the kernel correctly: it is here that the axioms `dia_setup` and `firewall_blackens` are used. The first few steps introduce the `dia_setup` axiom and instantiate it appropriately.

```
   (lemma "dia_setup")
   (grind :if-match nil)
   (apply (repeat (inst? -1)))
```

The reason for not using `grind-with-lemmas` to do this is that we need to leave premise -5 uninstantiated.

```
firewall_works.4 :

{-1}  dia(a!1)(b!1) AND segs(B)(a!1) AND
          segs(current(s!1))(b!1) AND NOT (current(s!1) = B)
       => a!1 = outbox AND current(s!1) = F AND NOT segs(B)(b!1)
[-2]  dia(a!1)(b!1)
[-3]  segs(current(s!1))(b!1)
[-4]  segs(B)(a!1)
[-5]  FORALL a: segs(B)(a) => black(s!1, a)
  |-------
[1]   select(s!1, b!1) = select(blacken(s!1), b!1)
[2]   black(next(s!1), a!1)
```

From here, we just do a case split and discharge the first case with the axiom `firewall_blackens`, and the second with `blacken_select`.

```
   (ground)
   (("1" (grind-with-lemmas :lemmas ("firewall_blackens")))
    ("2" (grind-with-lemmas :lemmas ("blacken_select")))))))
```

This step completes the proof that `firewall_works.` PVS is able to tell us that the proof depends on the following axioms and definitions (it also documents the previously

proved theorems that are used—these are omitted here because they all come from the built in "prelude" library).

```
firewall.firewall_works has been PROVED.

  The proof chain for firewall_works is COMPLETE.

  firewall_works depends on the following axioms:
    firewall.blacken_select
    firewall.firewall_blackens
    firewall.blacken_blackens
    firewall.blacken_current
    firewall.dia_setup
    firewall.black_funof_seg
    sk.separation
    firewall.spontaneous_generation

  firewall_works depends on the following definitions:
    reals.>=
    reals.<=
    sets.intersection
    relations.symmetric?
    relations.equivalence?
    sets.member
    relations.transitive?
    firewall.allblack
    relations.reflexive?
    sk.eqvals
    notequal./=
    sk.run
    firewall.blackset
    sets.fullset
```

As with the basic `kernel` theory, we should be concerned that the axioms in the `firewall` may be inconsistent. As before, we allay this concern by exhibiting an interpretation on to a constructively-defined theory. Our interpretation builds on that used before: we import the `kernel_consistency` theory and use the `kinst` defined there as the interpretation of `sk` within an interpretation of the `firewall` theory.

```
firewall_consistency: THEORY
BEGIN

 IMPORTING kernel_consistency

 IMPORTING firewall{{
  sk := kinst,
  black(s:mstates, a:addrs) := s`mem(a)=0,
  blacken(s:mstates) := s WITH [mem := LAMBDA(a:addrs): 0],
  F:=1,
  B:=2,
  outbox:=10
}}

END firewall_consistency
```

Our interpretation uses the value `0` to represent "black" values; thus the predicate `black` simply checks that the value of its segment is `0` in the state concerned, and the function `blacken` assigns `0` to all `addrs` in the state concerned (the `WITH` construction indicates function overriding: here the `mem` field of `s` is replaced by the function that is everywhere `0`). Observe that here we have specified the interpretations of `black` and `blackens` in an "applicative" style, rather than using explicit `LAMBDA` terms as we did in the previous interpretation. The applied form is superficially more attractive but requires care: the predicate or function being interpreted (e.g., `black`) is from the "upper" theory, but the types appearing in its argument list (e.g., `mstates` and `addrs`) are from the interpreting or "lower" theory.

Typechecking this theory generates the TCCs necessary to ensure that the seven axioms of the `firewall` theory become provable theorems under the translation specified by `firewall_consistency.` All of these TCCs are discharged automatically using the default proof strategy.

```
% Mapped-axiom TCC generated (at line 151, column 11)
  % proved - complete
IMP_firewall_dia_setup_TCC1: OBLIGATION
  FORALL (a, b: addrs, p: addrs):
    member[addrs](a, singleton[addrs](b)) AND
     member[addrs](b, emptyset[addrs]) AND
      member[addrs](a, emptyset[addrs]) AND p /= 2
     => b = 10 AND p = 1
           AND NOT member[addrs](a, emptyset[addrs]);

% Mapped-axiom TCC generated (at line 151, column 11)
  % proved - complete
IMP_firewall_firewall_blackens_TCC1: OBLIGATION
  FORALL (s: mstates): s‘pc = 1
     AND s‘mem(10) = 0 => id[mstates](s)‘mem(10) = 0;

% Mapped-axiom TCC generated (at line 151, column 11)
  % proved - complete
IMP_firewall_spontaneous_generation_TCC1: OBLIGATION
  FORALL (s: mstates): allblack(s) => allblack(id[mstates](s));

% Mapped-axiom TCC generated (at line 151, column 11)
  % proved - complete
IMP_firewall_black_funof_seg_TCC1: OBLIGATION
  FORALL (a: addrs, s, t: mstates):
    s‘mem(a) = t‘mem(a) => s‘mem(a) = 0 = (t‘mem(a) = 0);

% Mapped-axiom TCC generated (at line 151, column 11)
  % proved - complete
IMP_firewall_blacken_current_TCC1: OBLIGATION
  FORALL (s: mstates): s‘pc = s‘pc;

% Mapped-axiom TCC generated (at line 151, column 11)
  % proved - complete
IMP_firewall_blacken_blackens_TCC1: OBLIGATION
  FORALL (s: mstates):
    allblack(s WITH [mem := LAMBDA (a: addrs): 0]);

% Mapped-axiom TCC generated (at line 151, column 11)
  % proved - complete
IMP_firewall_blacken_select_TCC1: OBLIGATION
  FORALL (a: addrs, s: mstates): s‘mem(a) = 0 => s‘mem(a) = 0;
```

We have now completed our development of the `firewall` application. The proofs of all the theorems, TCCs, and lemmas in this and the `kernel` specification (15 proofs in total) can be rerun automatically in 2.12 seconds on a 2 GHz Pentium. Of course, human

time is more important than computer time; the entire development reported here took less than a day.

4 Conclusion

We have replicated in PVS the development and examination of a formal security policy for a separation kernel originally performed in ACL2 by Greve, Wilding, and Vanfleet [GWV03].

Although PVS and ACL2 are broadly comparable in the power of their theorem provers, they differ significantly in their other resources. In particular, the specification language of PVS is a higher-order logic, whereas that of ACL2 is a form of the programming language Lisp. Thus, the PVS specifications for both the separation policy and the firewall application are more abstract then their ACL2 equivalents: the PVS specifications use mathematical concepts such as types, sets, and quantification, while the ACL2 specifications are more operational and use programming concepts such as lists and functions defined over them.

The PVS proofs are fairly direct, involving little more than case analysis and selection and instantiation of appropriate axioms. Because they are developed interactively, the user indicates directly where to invoke each axiom (and sometimes how to instantiate it). The ACL2 proofs, on the other hand, are fully automatic, and user guidance is provided rather indirectly, as indicated by the following fragment.

```
(defthm intersection-equal-dia-b-segs-f-helper
  (implies
   (and
    (member x (segs 'b))
    (not (equal x 'outbox))
    (subsetp z (dia x)))
   (equal (intersection-equal z (segs 'f)) nil))
  :hints
  (("Subgoal *1/3'4'"
    :use (:instance dia-setup (seg1 z1) (seg2 x) (p 'f))))
  :rule-classes nil)
```

This `defthm` is concerned with the intersection of two sets, represented as lists. Due to the list representation, proofs require induction, and this is a "helper" lemma that is needed to make the sequence of inductive proofs successful. The `:hints` and `:rule-classes` annotations are needed to nudge the theorem prover in the right direction. The ACL2 proof contains several similarly annotated helper functions whose development must require considerable skill.

Consistency is demonstrated in PVS using theory interpretations, and this mechanism is able to deal with theories that combine axiomatic and constructive styles of specification. In ACL2, on the other hand, fully axiomatic constructions are supported by the `encapsulates` mechanisms, but there is no automated support for mixed presentations.

Thus, in the ACL2 development, demonstration that the axioms introduced for the firewall (corresponding to `dia_setup` and `firewall_blackens` in the PVS presentation) are consistent with the rest of the presentation requires what a comment in the ACL2 file `make-consistency-test.lisp` describes as "sneaky coding."

Although it is used here only for demonstration of consistency, PVS support for theory interpretations can also be used to establish correctness of hierarchical developments of increasingly concrete specifications. Thus, although the starting point of the PVS treatment is a highly abstract formulation of the separation security policy, it would be very natural to refine this and the firewall specification into a concrete implementation, with correctness proved at each step using theory interpretations.

This example demonstrates how the linguistic resources of PVS allow perspicuous and succinct specification of an abstract security policy and of a more concrete application that uses that policy. The language and other resources of PVS also allow straightforward demonstration of consistency for the axioms employed in these specifications. The expressiveness of the PVS language does not reduce proof efficiency—quite the reverse, because concepts such as predicate subtypes actually aid the theorem prover by making it easy to locate relevant facts. Certainly, the proofs involved in this exercise were simple to construct and support the "computational Sapir-Whorf hypothesis" posited by Shankar [Sha01] that language expressiveness positively influences proof efficiency.

References

[GWV03] David Greve, Matthew Wilding, and W. Mark Vanfleet. A separation kernel formal security policy. In *Fourth International Workshop on the ACL2 Theorem Prover*, Boulder, CO, July 2003. Available at http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/#presentations. i, 1, 8, 17

[KMM00] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*, volume 3 of *Advances in Formal Methods*. Kluwer, 2000. 1

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995. 1

[OS01] Sam Owre and N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2001. 6

[OSRSC98] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1998. Three volumes: Language, System, and Prover Reference Manuals. 1

[Rus81] John Rushby. The design and verification of secure systems. In *Eighth ACM Symposium on Operating System Principles*, pages 12–21, Asilomar, CA, December 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5). 1

[Sha01] Natarajan Shankar. Using decision procedures with a higher-order logic. In *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 5–26, Edinburgh, Scotland, September 2001. Springer-Verlag. Available at ftp://ftp.csl.sri.com/pub/users/shankar/tphols2001.ps.gz. 18

[Sho67] Joseph R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, MA, 1967. 6