Using composition and refinement to support security architecture trade-off analysis

C. N. Payne, Jr.
Secure Computing Corporation
Roseville, MN USA
cpayne@securecomputing.com

Abstract

This paper demonstrates that composition and refinement techniques are a promising solution for performing rigorous, security architecture trade-off analysis. Such analysis typically occurs in one of two forms: comparing two architectures for implementation and determining the impact of change to an implemented architecture. Composition and refinement techniques reduce the overhead of such analysis significantly over traditional formal methods by facilitating specification and proof reuse and by providing powerful reasoning tools.

In this paper, we propose an approach for applying composition and refinement techniques to trade-off analysis. Our approach relies on a formal composition and refinement framework, which is not described here. We describe the approach and apply it to a simple example. We conclude with lessons learned and future work.

1 Introduction

Security engineers routinely face the following questions:

- Given two or more candidate security architectures, which is better?
- How will modifying subsystem x affect my system's security posture?

Both questions require some sort of *trade-off analysis*. In the first case, the engineer compares two proposed architectures and trades off the strengths and weaknesses of one against the other. In the second case, the engineer trades off the strengths and weaknesses of the proposed changes against the strengths and weaknesses of the existing implementation. The winner in both cases offers the best overall balance of concerns.

Several factors weigh into a security architecture trade-off analysis. A critical factor is the security vulnerability assessment. To make a competent decision, the engineer must identify and understand the potential vulnerabilities of the proposed architecture. Another factor is the way in which the architecture satisfies its security obligations. This contributes to the determination of assurability. Other factors include cost, ease of maintenance, etc.

Most factors are established informally, but others, such as the vulnerability assessment, require a rigorous investigation. Payne, Froscher and Landwehr [6] propose an approach based on assumptions and assertions that offers an informal, systematic exploration for vulnerabilities. For each system component, assertions specify how the component will behave, and assumptions specify what the component expects from its environment. Vulnerabilities occur within a component when environmental assertions fail to validate the component's assumptions. The approach extends the investigation beyond the technical boundary of a computer system to include assumptions and assertions about physical security, personnel security, communications security and so on. The engineer can use the assumptions/assertions approach to answer questions such as *how does each candidate rely on the rest of the system?* and *what are the effects of change?*

Although the process of matching assumptions against assertions is systematic, the assumptions/assertions approach does not attempt to verify the accuracy of the assertions and assumptions themselves. In other words, it cannot answer with certainty the question: *does the component satisfy its security obligations?* If the trade-off analysis demands that level of rigor, then we must rely on other tools. Formal methods are a natural choice, but traditional tools require significant reasoning support to be practical for trade-off analysis. In particular, the analysis of two similar architectures demands support for specification and proof reuse.

This paper demonstrates that formal composition and refinement techniques, such as described by Abadi and Lamport [1, 4] and by Shankar [8], can support cost-efficient,

rigorous security architecture trade-off analysis.[1] The techniques facilitate reuse of specifications and proof, which reduces the overhead of evaluating similar candidates. They are component-based, that is all reasoning occurs at the granularity of the component. A component can be "swapped out" as long as its replacement satisfies the same requirements. This supports ease of change. The techniques offer powerful reasoning tools for reducing a problem to manageable pieces. Finally, their formal foundation lets the impact of change be realized quickly, i.e., proofs break.

We propose an approach for applying composition and refinement techniques to security architecture trade-off analysis. The success of our approach relies on a formal composition and refinement framework, which we will not describe in detail here. The framework is derived from Fine [2] and is based heavily on the work of Abadi-Lamport [1, 4] and Shankar [8]. It is implemented in PVS [5] and consists of about two dozen theories.

Section 2 describes the primary reasoning tools provided by composition and refinement. Section 3 discusses how these tools are used in our approach. Section 4 demonstrates the approach on a simple example.

2 Composition and refinement

Composition and refinement are complementary approaches for demonstrating that a system satisfies its critical properties.

Composition supports bottom-up reasoning about the system in the same way that modular software design supports its bottom-up construction. Composition techniques let us model the system by combining the models of its components. To reason about the system, we show that the component properties together imply the property desired for the system. The technique supports "plug-and-play": each component can be replaced as long as its replacement satisfies the component's properties. In addition, since the component analysis depends only on that component, the analysis is reusable when the component is used in a new context.

Refinement, on the other hand, supports top-down reasoning. We start by defining an abstract model of the system and demonstrating that it satisfies certain properties. We then refine the abstract model — by adding more detail — until we reach a model that corresponds more closely to the implementation. It is usually easier to prove desired system properties of the abstract model than for a more detailed one; on the other hand, it is easier to relate the more detailed model to the implementation since it includes more design details. Refinement techniques let us show that properties demonstrated for the abstract model are preserved in its refinements. Thus, we only need to analyze the property at the abstract level where the analysis is easiest.

Combining composition and refinement, then, provides a powerful reasoning tool for complex systems.

The following sections describe the basic building block, the *component*, and tools for reasoning about composition, property satisfaction and refinement.

2.1 Component

Abadi-Lamport specify a component using the following normal form

$$\exists x : I \wedge \Box[\mathcal{N}]_v \wedge F \tag{1}$$

where

- x denotes variables that are internal to the component.
- I is a state predicate characterizing the initial state,
- $\mathcal{N}$ is an action predicate characterizing valid state transitions,
- v characterizes the "protected" state information that may be changed only by $\mathcal{N}$-transitions,
- $[\mathcal{N}]_v$ denotes the set of transitions that are either an $\mathcal{N}$-transition or in which the state described by v remains unchanged,
- $\Box[\mathcal{N}]_v$ means all state transitions are in $[\mathcal{N}]_v$, and
- F is a fairness condition that is the conjunction of "weak" and "strong" fairness conditions on steps comprising $\mathcal{N}$.

In order to reason about their composition or refinement, all components must be defined on a common state. The preferred approach is to define a state structure that includes an access function for each component's state. The component is then defined in terms of its access function. This approach is preferred because it allows new components to be added to the common state without affecting the analysis of existing components.

2.2 Composition

The basic idea of composition is

- the composed components start in a common state that is acceptable to all of them,
- the components take turns performing transitions, and
- the fairness conditions of all the components are satisfied.

[1] This research was supported by the Maryland Procurement Office under Contract MDA904-97-C-3047.

Abadi-Lamport define composition as simply conjunction. Ignoring quantification, the composition of components 1 through n is defined as

$$\bigwedge_{i=1}^{n} I_i \wedge \Box[\bigcup_{i=1}^{n} \mathcal{N}_i]_{\langle v_1,\ldots,v_n\rangle} \wedge \bigwedge_{i=1}^{n} F_i \qquad (2)$$

under the assumptions that $\mathcal{N}_i \Rightarrow v_j' = v_j, \forall i,j : i \neq j$. That is, each $\mathcal{N}_i$ must guarantee that the protected information of other components is unchanged. When defining a component, an analyst is not — and should not be — concerned with protecting the state information of components with which it might later be composed. Thus, a typical $\mathcal{N}$ may not satisfy these assumptions. In other words, it is possible to specify a composite in which some v_j is explicitly violated by $\mathcal{N}_i$. So $\mathcal{N}$ must be augmented by hand during composition.[2]

We use *compose*(S) to denote the composition of components in set S. A composite system is itself considered a component.

2.3 Satisfaction

A component specification defines a set of *behaviors*, or sequences of transitions [1]. The term *mprop*(c) denotes the set of behaviors for component c such that each behavior starts in an initial state for c, engages in transitions for c and satisfies the fairness conditions. The *mprop* of a composite is the intersection of the *mprop*s of its components.

A property is merely a set of behaviors. We say that component c satisfies some desired property P, written *satisfies*(c, P), if all of the behaviors in *mprop*(c) occur in P. This is called *unconditional* satisfaction. If c is a composite, we can break the proof into a subproof for each of the components.

We can also perform *conditional* satisfaction: a component satisfies P only if certain assumptions hold, written *assum_satisfies*(c, P). In this case, we prove that c satisfies P given the its assumptions. If c is a composite, we can reduce the proof to showing that one of c's components conditionally satisfies P and that the assumptions of that component are justified by all components with which it is composed. Conditional satisfaction results can be reused when the component is composed with different peers. Only the justification of assumptions must be redone.

2.4 Refinement

Component a is a refinement of, or *implements*, component b (written *implements*(a, b)) if *mprop*$(a) \subseteq$ *mprop*(b). Refinement preserves satisfaction. If *satisfies*(b, P) and *implements*(a, b) then *satisfies*(a, P). *implements* is also transitive.

Typically, either a or b (or both) is a composite, so tools are provided to decompose the proof of *implements*(a, b) into manageable pieces. There are four cases of refinement decomposition:

1. *1 – 1* — In the simplest case, where neither a nor b is a composite, we can reduce the proof to three cases:
   - every initial state of a is an initial state of b,
   - every transition for a is allowed by b, and
   - every behavior in *mprop*(a) satisfies the fairness conditions for b.
2. *many – 1* — If a is a composite but b is not, then we treat like the *1 – 1* case, except that the second step is rewritten as: the transition must be performed by some component in *compose*(S) and allowed by all other components in *compose*(S), where $a =$ *compose*(S).
3. *many – many* — If both are composites, the proof should be decomposed into subproofs of types *1 – 1* and *many – 1*. This is done by defining a set of component pairs — each pair consisting of a specification component and an (possibly composite) implementation component — and reducing the main implementation proof to a subproof for each component pair. If each subproof depends entirely on its own component pair (a pure decomposition), we isolate each subproof for quick resolution. If a subproof relies on other components (an impure decomposition), we break the main implementation proof into a set of basic obligations, which are derived from Abadi-Lamport's Decomposition Theorem [1]. An impure decomposition is sometimes necessary if the granularity of transitions changes from one specification level to the next.
4. *1 – many* — This is an unusual case, but if the refinement is correct, it may be treated as a pure decomposition.

It is beneficial to combine refinement analysis and compositional analysis. For example, if the abstract component is a composite for which conditional analysis has been performed, we can demonstrate that the conditional properties are satisfied by the composite and that the composite satisfies some critical property. Since the property is true of the abstract component, it will also be true of any refinement. Furthermore, if one component is swapped out for another, the affected analysis is very localized. Much of the critical property proof and refinement proof can be reused.

[2] The framework we used overcomes this limitation.

3 Approach

Our approach for applying composition and refinement techniques to security architecture trade-off analysis is very simple. We consider each trade-off scenario from Section 1 in turn.

Given two or more candidate architectures, which candidate is better? Our goal is to expose the security assumptions and assertions for each candidate by trying to prove that the candidate satisfies the security obligations imposed on it. Proving this of each candidate would be time-consuming and possibly redundant, since the candidates may be similar. Also, depending on the level of detail, the proof may be very difficult.

Instead we propose an abstract model that represents all candidates and prove that the model satisfies the requirements. Then we prove that each candidate is a refinement of the abstract model. If we are unable to complete a refinement proof, it may be that the abstract model does not reflect that candidate, or it may be that the candidate will not satisfy the security requirements. Only further analysis will tell. However, for each candidate successfully refined, refinement theory lets us claim that the candidate therefore satisfies the security requirements. The differences between candidates, then, exist in the assumptions and assertions for each candidate that are necessary to demonstrate refinement. At this point, the engineer can weigh the assumptions and assertions against some other criteria (e.g., ease of change).

Our approach has several benefits. Refinement proofs are usually easier than satisfaction proofs, and if the candidates *are* similar, part of the refinement proof may be reused. If a new candidate is later proposed, we need only prove that it is a refinement of the abstract model. If a candidate is later modified, only the refinement proof must be redone.

If a candidate is complex, we may use composition tools to build it up from its parts. Similarly, we can use decomposition tools to break a candidate into smaller pieces that reflect its intended implementation. The decomposition proof will distribute the requirements on the candidate to requirements on each component of the candidate.

How will modifying subsystem x *affect my system's security posture?* If x was previously analyzed as described above, we only need to introduce the changes to x and see if its refinement proofs are still valid. If it was not, or if the modifications introduce new security requirements, we will follow the approach described above, i.e., propose an abstract model, prove the security requirements, prove refinement, etc.

Composition and refinement techniques help us rigorously identify the security assumptions and assertions that will be used in the trade-off analysis. Reasoning tools let us quickly — and we suspect, automatically — reduce a refinement proof in a way that corresponds to our intuitive view of refinement, i.e., that the requirements of a more detailed component imply the requirements of a more abstract component. In addition, once the reduction occurs, the requirements of the more detailed component can be modified as necessary to complete the refinement — with little impact to previous analysis. This supports rapid change.

4 Example

Now we will demonstrate our approach on a simple example. In the interest of space, our demonstration considers only unconditional satisfaction[3] and pure refinement decomposition. In addition, we reduce the scope of our effort by ignoring fairness conditions. For convenience, we denote I and $[\mathcal{N}]_v$ in the component state as *init* and *steps*, respectively.

Our example is drawn from the research of secure, microkernel-based, operating system architectures, such as Fluke [3]. Consider a simple, generic (i.e., non-security-relevant) system consisting of three object managers (ignore the kernel): a file server, a process manager and a memory manager. The system is illustrated in Figure 1.

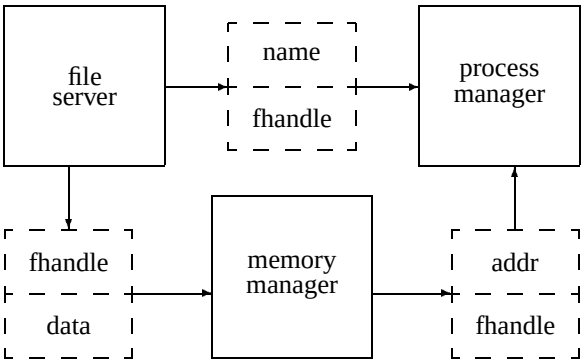

Figure 1. Simple system

The file server manages the file system and handles file requests from the process manager and the memory manager. The process manager maintains all running processes. The memory manager provides address spaces to the process manager for storing process executables. The interfaces provided in Figure 1 are simple to reflect the critical security property that we will introduce below. When the process manager needs a new executable for a process, it queries the file server using the `name` of the file that it (the process

[3] Some satisfaction proofs described below (e.g., the obligations on `pm_comp`) are actually conditional; however, we make the assumptions explicit in the theorem to be proved, e.g., *satisfies*$(c, (A \Rightarrow P))$ where A is an assumption. This tactic does not support reuse, but it simplifies the demonstration!

manager) knows contains the executable. The file server responds with a file handle `fhandle`. The process manager then queries the memory manager for an address space containing the contents of the file referenced by `fhandle`. If necessary, the memory manager queries the file server with `fhandle`, and the file server responds with `data`, which represents the contents of the file. The memory manager creates a new address space, dumps the contents in it, and returns a pointer to the address space, `addr`, to the process manager. The process manager then assigns the value of `addr` to the new process.

Following a strategy of specifying as little as possible, we model communication in one direction only (noted by the arrows in Figure 1). Thus, for example, it appears that the file server randomly places a `fhandle` and its corresponding `name` on the interface with the process manager. This strategy simplifies the requirements and lets us isolate the file server from the memory manager and the process manager since, according to its interface, the file server depends on no other component. This fact will prove useful in the refinement, because we can isolate the file server analysis and reuse it.

Now we modify the architecture to support the following critical property:

> (Critical Property) All trusted processes have assured executables.

In other words, all processes that are trusted to perform critical functions in the system run with executables that have been examined against some rigorous standard for correctness and security. Two candidates are proposed. Both candidates trust the file server to provide assured executables upon request. The difference between the candidates is whether the process manager or the memory manager is more highly trusted to assign executables to processes correctly.

In the first candidate architecture, illustrated in Figure 2, the process manager knows which processes are trusted. When it needs to assign a new executable to an existing trusted process, it prompts the file server with the `name` as described above. The file server responds with the file handle `fhandle`. The file server also notes, by setting the `trusted?` flag, whether the contents of the file are considered assured.

The rest of the operation is performed like in the generic system above. Note that while the memory manager handles assured data, it does not know the data is assured. Assured data could become unassured by changing it. To ensure that the process manager assigns the same contents that the file server intended, we add the requirement that the memory manager cannot change the contents of an address space.

In the second candidate architecture, illustrated in Figure 3, the process manager does *not* know about trusted

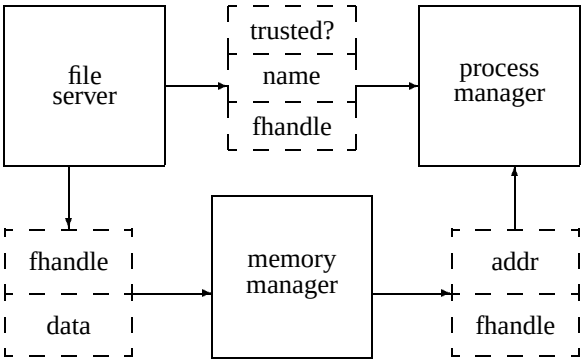

Figure 2. Candidate 1

processes. It makes the same request of the file server, but this time the file server responds like in the generic system, i.e., only with the file handle `fhandle`. The file server does not indicate whether that file handle represents assured file contents. The process manager's query to the memory manager is also the same, and the memory manager makes the same query to the file server. This time, however, the file server indicates for the memory manager, via `trusted?`, whether assured contents are stored in `data`. If `data` is assured, the memory manager creates a *trusted address space*, places the contents of `data` in it, and returns the pointer to it, `addr`, to the process manager. Trusted processes, then, correspond to trusted address spaces.

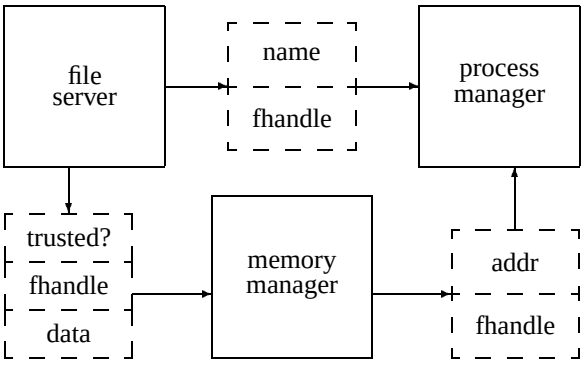

Figure 3. Candidate 2

To prove that each candidate satisfies the critical property would be time-consuming and would result in a duplication of effort since the obligations on the file server do not change. So we propose an abstract model (see Figure 4) that retains the file server but combines the process manager and memory manager into a single process manager. We can abstract away file handles and address spaces without loss of reason-

ing. We specify the file server component `fs_comp` and the process manager component `pm_comp`.

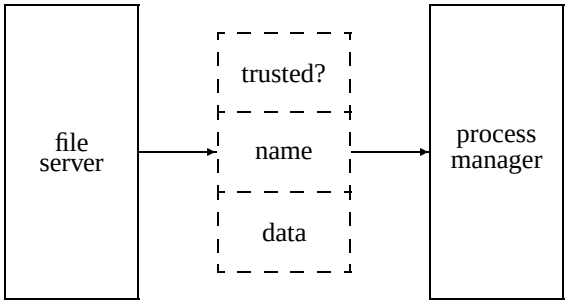

Figure 4. Abstract model

The first step in the proof that the abstract model satisfies the critical property is to identify the proof obligations for each component. In keeping with the strategy for specifying component communication, only `fs_comp` can change the interface. Therefore, `pm_comp` relies on the interface being set correctly.

We have not yet discussed how `fs_comp` knows which file contents are assured. The meaning of "assured" is not specified — it could be a tag assigned by some external user. `fs_comp` can only recognize assured data by where the data is stored. It knows about two types of files: ordinary files and *trusted files*. Trusted files are assigned only assured contents. When `fs_comp` sets the `trusted?` flag on the interface, it is indicating that `data` came from a trusted file. Therefore, we claim the following obligations for `fs_comp`:

- All trusted files always have assured contents.
- Whenever the `trusted?` flag is set, the value of `data` is assured.

The obligations for `pm_comp` follows:

- Given that the obligations for `fs_comp` are met, all trusted processes always have assured executables.

We compose `fs_comp` and `pm_comp` into a single component called `top_level` and assert

$$\textit{satisfies}(\texttt{top\_level}, \text{Critical property}),$$

then we demonstrate that the obligations for `fs_comp` and `pm_comp` are sufficient to satisfy this expression. Next we assert and prove component requirements to satisfy each obligation. We verify that the following requirements are sufficient to satisfy the `fs_comp` obligation:

1. (*init*) In the initial state,
   (a) `name` is a valid file, and `data` is its contents.
   (b) If the `trusted?` flag is set, `name` is a trusted file.
2. (*steps*) For every state transition,
   (a) If the interface changes, 1a holds.
   (b) If the interface changes, 1b holds.
   (c) If a trusted file is created, its contents are assured.
   (d) If a trusted file is modified, the new contents are assured.

We verify that the following requirements are sufficient to prove the `pm_comp` obligation:

1. (*init*) In the initial state,
   (a) All trusted processes have an assured executable. (This is a "bootstrapping" requirement.)
2. (*steps*) For every state transition,
   (a) If the executable for a process changes, the new executable is assigned from `data` as long as `name` denotes the appropriate executable filename for this process.
   (b) If the executable for a trusted process changes, the above holds *and* the `trusted?` flag must have been set.
   (c) If a new process is created, it is copied from an existing process.
   (d) If a new trusted process is created, it is copied from an existing trusted process.

Any implementation that satisfies these requirements will also satisfy the critical property.

Now we consider each refinement in turn. For the first candidate, we specify the refined file server `R1_fs_comp`, the refined process manager `R1_pm_comp` and the memory manager `R1_mm_comp`. Then we compose `R1_pm_comp` and `R1_mm_comp` into a single component called `R1_mm_pm`, and then compose `R1_mm_pm` with `R1_fs_comp` to form the component `R1_level`. We assert

$$\textit{implements}(\texttt{R1\_level}, \texttt{top\_level})$$

and use the pure decomposition tools to reduce the above expression to:

- $\textit{implements}(\texttt{R1\_fs\_comp}, \texttt{fs\_comp})$, and
- $\textit{implements}(\texttt{R1\_mm\_pm}, \texttt{pm\_comp})$

We reduce each of these to obligations for `R1_fs_comp` and obligations for `R1_mm_pm`, respectively. We show that given a refinement mapping,

- *init*(R1_fs_comp) ⇒ *init*(fs_comp)
- *steps*(R1_fs_comp) ⇒ *steps*(fs_comp)

and

- *init*(R1_mm_comp) ∧ *init*(R1_pm_comp) ⇒ *init*(pm_comp)
- *steps*(R1_mm_comp) ∧ *steps*(R1_pm_comp) ⇒ *steps*(pm_comp)

where *init*(c) and *steps*(c) are sets of requirements defining the allowed initial state and the allowed transitions for component c, respectively. In the interest of space, we will not list those requirements or the refinement mapping here. The complete specification is available in [7]. The result is a set of requirements that represents assumptions and assertions for R1_fs_comp, R1_pm_comp and R1_mm_comp. An interesting side effect is that we can modify these requirements and the associated refinement mapping as necessary to complete the obligation proof *without* affecting the refinement proof reduction to this point, because the refinement reduction does not rely on those requirements.

The refinement for the second candidate is very similar. The main difference is that it shares R1_fs_comp from the first refinement. As a result, we get for free all proof results for R1_fs_comp.

5 Summary

We have described an approach for applying composition and refinement techniques in support of rigorous, security architecture trade-off analysis, and we have demonstrated the approach on a simple example.

We actually performed the analysis described in the example using a composition and refinement framework that we developed in PVS.[4] We successfully proved *implements*(R1_fs_comp, fs_comp), which applies to both candidates. We also completed the refinement reduction proof for both candidates. Unfortunately, we ran out of time before confirming that the assertions and assumptions that we proposed for the process manager and the memory manager in each candidate are sufficient. However, as we noted, the requirements can be manipulated to complete these proofs without affecting earlier analysis.

Acknowledgements

The author thanks Todd Fine, Ed Schneider, Duane Olawsky and Tom Sundquist, whose combined efforts provided the foundation for the work described here. Ray Spencer provided valuable comments on earlier drafts. Thanks also to Pete Loscocco and Steve Smalley for helpful suggestions.

[4] The framework and its supporting documentation are available at http://www.securecomputing.com/css.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. Technical Report 118, Digital Equipment Corporation, Systems Research Center, Dec. 1993.

[2] T. Fine. A Framework for Composition. In *Proceedings of the Eleventh Annual Conference on Computer Assurance* , pages 199–212, Gaithersburg, Maryland, June 1996.

[3] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Symposium on Operating Systems Design and Implementations (OSDI)*, Oct. 1996.

[4] L. Lamport. The Temporal Logic of Actions. *ACM Trans. Prog. Lang. Syst.*, 16(3):872–923, May 1994.

[5] S. Owre, N. Shankar, and J. Rushby. *The PVS Specification Language*. Computer Science Laboratory, SRI International, Menlo Park, CA 94025.

[6] C. N. Payne, J. N. Froscher, and C. E. Landwehr. Toward a comprehensive INFOSEC certification methodology. In *Proceedings of the 16th National Computer Security Conference* , pages 165–172, Baltimore, MD, September 1993. NIST/NSA.

[7] Secure Computing Corporation. Assurance in the Fluke Microkernel: System Composition Study Report. CDRL A005, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, 1998. In preparation.

[8] N. Shankar. A lazy approach to compositional verification. Technical Report TSL-93-08, SRI International, Dec. 1993.