Beyond Traces and Independence*

Fred B. Schneider
Department of Computer Science,
Cornell University,
Ithaca, New York 14853 USA
fbs@cs.cornell.edu

Abstract. The formal methods, fault-tolerance, and cyber-security research communities explore models that differ from each other. The differences frustrate efforts at cross-community collaboration. Moreover, ignorance about these differences means the status quo is likely to persist. This paper discusses two of the key differences: (i) the trace-based semantic foundation for formal methods and (ii) the implicit notions of independence.

1 Introduction

Computing systems we depend on should do what we expect and nothing more. That challenge is being tackled today by researchers in three communities.

− The formal methods community studies means for gaining assurance in the properties that a given system satisfies when executed in some prescribed (often idealized) execution environment.
− The fault-tolerance community focuses on algorithms and system architectures for tolerating various kinds of natural events that disrupt the execution environment.
− The cyber-security community worries about designing defenses to resist attacks intended to circumvent system controls and compromise system operation.

This paper explores important differences in the models each community studies. Because of these differences, results developed by one community are not necessarily applicable to the questions studied by the others. Moreover, differing implicit assumptions in the models make it difficult even to recognize when results from one community can be applied to matters of concern by another.

Needless to say, incompatibilities in the different models studied by the three research communities only frustrate efforts to understand how we might go about

* Supported in part by National Science Foundation grants 0430161, 0964409, and CCF-0424422 (TRUST), ONR grants N00014-01-1-0968 and N00014-09-1-0652, and a grant from Microsoft. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.
building systems on which we can depend. Additional research can reconcile those differences, ultimately bridging the gaps between the three closely related areas. This paper is intended to inform and inspire that endeavor.

2 Formal Methods

Research in formal methods concerns the development and use of programming logics and model checkers to gain confidence about what behaviors a system can and cannot exhibit. By choosing trace properties (defined below) as a foundation, formal methods researchers obtain elegant characterizations for whether a program satisfies a specification and they can support compositional development as well as step-wise refinement of programs. So a considerable body of formal methods research adopts trace properties or some other foundation having roughly equivalent expressive power.

Trace properties, however, (as is shown in §2.2) are inadequately expressive for specifying security, where requirements are typically described in terms of the following elements.

Confidentiality. Which principals are allowed to learn what information.

Integrity. What changes to the system (stored information and resource usage) and to its environment (outputs) are allowed.

Availability. When must inputs be read or outputs produced.

The choice of trace properties as the foundation for a formal method thus creates a gap between the kinds of system behavior we can reason about and the defining elements of security.

A generalization of trace properties—hyperproperties [3]—is sufficiently expressive, but programming logics and model checkers have not (yet) been developed for this foundation. And although hyperproperties might not turn out to be the right foundation, it is clear that something significantly more expressive than trace models is needed to support what the security community needs.

2.1 Trace Properties

A trace is a (possibly infinite) sequence; a trace property is a set of traces, where each trace in isolation satisfies the characteristic predicate associated with that trace property.[1] Examples of trace properties include partial correctness (the first state satisfies the input specification and any terminal state satisfies the output specification), mutual exclusion (in each state, the program for at most one process designates an instruction in a critical section), and termination (at some point in the trace, there is a terminal state and thereafter it is repeated).

[1] For concreteness, we consider traces that are sequences of states. Similar arguments can be made if traces are sequences of actions.