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 is that something significantly more expressive than trace models is needed to 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.

Lamport [6] provides an intuitive classification of trace properties into *safety* and *liveness*. Safety asserts that no “bad thing” happens during execution, and liveness asserts that some “good thing” happens. For example, mutual exclusion is safety (the “bad thing” is a state where both processes are executing in critical sections) and termination is liveness (the “good thing” is an infinite suffix of terminal states).

Lamport’s classification can be formalized to prove that every trace property is either safety, liveness, or the conjunction of two trace properties—one that is safety and one that is liveness [1]. In addition, an invariance argument suffices for proving that a program satisfies a trace property that is safety; a variant function is needed for proving a trace property that is liveness [2]. Thus, the safety-liveness classification for trace properties comes with proof methods beyond offering formal definitions. This suggests that the classification adds value beyond providing a taxonomy, because the classification provides insights into what methods must be used for reasoning about a given property.

A program S can be modeled as a trace property Σ_S containing all traces that could arise from executing S, and a specific execution of S *satisfies* a trace property P, if and only if the trace σ modeling that execution satisfies $\sigma \in P$. Thus, a program S *satisfies* a trace property P denoted $S \models P$ if and only if $\Sigma_S \subseteq P$ holds. In addition, we say that a program S' *refines* S, denoted $S' \preceq S$, when S' resolves choices left unspecified by S. For example, a program that increments x by 1 refines a program that merely specifies that x be increased. A refinement S' of S thus exhibits a subset of the executions for S: $S' \preceq S$ holds if and only if $\Sigma_{S'} \subseteq \Sigma_S$ holds.

Notice that $\models$ is closed under refinement. That is, if $S' \preceq S$ holds and $S \models P$ holds, then $S' \models P$ necessarily holds. This is because $S' \preceq S$ implies $\Sigma_{S'} \subseteq \Sigma_S$, so from $\Sigma_S \subseteq P$ (due to $S \models P$) we conclude by transitivity of $\subseteq$ that $\Sigma_{S'} \subseteq P$ holds. Also, if we construct S' by performing a series of refinements $S' \preceq S_1$, $S_1 \preceq S_2$, ..., $S_n \preceq S$ and S satisfies P then we are guaranteed that S' will satisfy P too. So programs that satisfy trace properties can be constructed by *step-wise refinement*.

2.2 Beyond Trace Properties

Now consider *information flow* between a variable x and a variable y, which is a security requirement (often prohibited for confidentiality) stipulating that the values of these two variables are correlated in all traces. We cannot tell whether a single trace in isolation satisfies information flow between x and y, because the values of these variables might be correlated by happenstance.[2] Only by considering all traces in Σ_S can we ascertain whether a program S satisfies the information flow requirement. And no characteristic predicate (on individual traces) can define exactly the set of execution traces where, for example, information flow does not occur. Thus, information flow and, consequently, its absence (confidentiality), cannot be formalized as trace properties. Some richer foundation is needed.

[2] We can tell that a single trace does not satisfy the information flow, however.

Unfortunately, once a more expressive foundation than trace properties has been adopted, "satisfies" is not necessarily closed under refinement. As an example, consider two programs. Program $S_{x=y}$ is modeled by trace property $\Sigma_{x=y}$ containing all traces in which $x = y$ holds in all states; program S^* is modeled by Σ_{S^*} containing all sequences of states. We have that $\Sigma_{x=y} \subset \Sigma_{S^*}$ holds, so by definition $S_{x=y} \preceq S^*$. However, program S^* "satisfies" the confidentiality requirement of no information flow between x and y, whereas (refinement) $S_{x=y}$ does not. "Satisfies" is thus not closed under refinement, and step-wise refinement is not sound for deriving programs whose requirements need a more-expressive formulation than trace properties provide.

We conclude that generalizing from trace properties to something more expressive will require revisiting and redeveloping the basic theory that today the formal methods community has good reason to hold dear. Otherwise, that foundation will keep the formal methods and cyber-security communities apart.

3 From Fault-Tolerance to Attack Tolerance

A system is considered *fault-tolerant* if it continues to operate correctly even when some of its components exhibit faulty behavior. Fault-tolerance is defined relative to a *fault model*, which specifies assumptions about what components can become faulty and specifies the behavior faulty components might exhibit. In the *Byzantine* fault model [7], faulty components may perform arbitrary state transitions and are even permitted to collude. A real system is unlikely to experience such hostile behavior from its faulty components, but any faulty behavior that might actually be experienced is, by definition, allowed with the Byzantine fault model. So by building a system that works assuming the Byzantine fault model, we ensure that the system can tolerate all behaviors that are actually exhibited by its faulty components.

The basic recipe for implementing such *Byzantine fault-tolerance* is well understood. It invariably employs redundancy in one form or another [8, §2.3]. For example, the state machine approach [10] applies to any system whose output is a function of the sequence of inputs it has received. With this approach, we (i) replicate the system $2t+1$ times and (ii) employ a protocol that ensures each of the (non-faulty) system replicas receive the same inputs in the same sequence. Provided that t or fewer replicas are faulty, then some majority subset of the $2t + 1$ will be correct and generate identical correct outputs. So the majority output from all replicas is unaffected by the behaviors of faulty replicas. We have built a t fault-tolerant version of the system.

Implicit in using this or any other construction involving replication is the assumption that replicas fail (approximately) independently. Under this *independence assumption*, the probability that a majority of the replicas is faulty is significantly smaller than the probability that fewer replicas are faulty. So the fault-tolerance of the overall system is better than the fault-tolerance of its individual components. However, if replicas exhibit correlated failures then replication does not enhance fault-tolerance. Fortunately, physically-separated components connected only by narrow-bandwidth channels are generally observed to exhibit uncorrelated failures and, therefore, approximate the independence assumption.

A faulty component in the Byzantine fault model is indistinguishable from a component that has been compromised and is under control of an attacker. We might thus conclude that if a Byzantine fault-tolerant system can tolerate t component failures then it also could resist as many as t attacks—we could get attack-tolerance by implementing Byzantine fault-tolerance. Unfortunately, the argument oversimplifies, and the conclusion is unsound. Physically-separated replicas share many of the same vulnerabilities (because they will use the same code) and, therefore, do not exhibit independence to attacks. If a single attack might cause any number of components to exhibit Byzantine behavior then little is gained by tolerating t Byzantine components. The security community thus cannot simply import Byzantine fault-tolerance methods from the fault-tolerance community.[3]

3.1 Obtaining and Preserving Independence

By eliminating all software bugs, we would eliminate the vulnerabilities that cause assumptions about replica-independence to be unsound when attackers are present. The construction of bug-free software is quite difficult, however. So instead we might turn to another means of increasing replica independence: diversity. Replicas need not actually be identical in either their design or their implementation—it suffices that different replicas produce equivalent responses for each given request. We use the term *morph* for this weaker notion of a replica.

The obvious way to create a set of morphs is by developing multiple implementations of each system component.[4] This, however, can be quite expensive, because the cost of all facets of system development are multiplied by the number of replicas to be developed. In addition, interoperation of diverse components is typically more difficult to orchestrate, not withstanding the adoption of standards. Finally, experiments have shown that distinct development groups working from a common specification will produce software having the same bugs [5].

Less costly is to employ a scheme that introduces diversity automatically during compilation, loading, or in the run-time environment [4,11]. We call this process *obfuscation*. Code can typically be generated and storage allocated in any number of ways for a given high-level language program. By making different choices in producing the executables for different replicas, we introduce a measure of diversity.[5] So, for obfuscation, morphs are produced by making

[3] In addition, confidentiality is not helped by the replication. Cryptographic solutions are required here—practical protocols for multiparty computation and homomorphic encryption would be ideal here.

[4] A special case is to procure pre-existing diverse components that have similar functionality or can be easily modified to implement the same interface.

[5] Different executables for the same high-level language program are still implementations of the same algorithms, though, so executables obtained in this manner will continue to share any flaws in those algorithms.

different choices for each executable. Moreover, the specific choices made in producing a morph are kept confidential (from attackers), whereas the program that performs obfuscation and the original source code are assumed to be public knowledge (hence known to attackers).

Diversity brings independence, hence the potential for attack tolerance. But this independence persists only as long as the cost to an attacker for successfully compromising a majority of the morphs is significantly larger than the cost of compromising a single morph. Unfortunately, actions by an attacker often can cheaply determine what transformations were performed to create each morph. That information then allows the attacker to design a custom exploit for each morph, if some vulnerability does exist in the source code from which the morph was derived. So independence of morphs produced using artificial diversity tends to erode over time.

One defense against such erosion of independence is *proactive obfuscation* [9]. Proactive obfuscation introduces epochs to system operation.

– In each epoch, some server is rebooted using a freshly generated, diverse executable.
– All n servers are rebooted after n epochs have elapsed.

Note that the inherent ability of a replicated system to tolerate outages by a minority of its components means that the server reboots at the end of each epoch can be made invisible to clients.

An epoch length of Δ seconds implies that an adversary is forced to compromise more than t replicas in $n\Delta$ seconds in order to subvert a service comprising n replicas. And we can make the compromise of more than t replicas ever more difficult for a resource-bounded adversary by reducing the length of Δ, although Δ is obviously bounded from below by the time needed to re-obfuscate and reboot a single server morph.

In summary, independence is clearly a crucial ingredient when replication is being used to tolerate aberrant behavior by components. But the means by which independence can be achieved differs in subtle ways, depending on whether natural events or attacks are to be masked. So the commonly held view—that attack-tolerance is merely a matter of choosing an appropriate fault model—misleads by ignoring independence. Independence will likely have to be elevated to being a first-class notion before the fault-tolerance and security communities could leverage each others' work on replication.

4 Discussion

History and sociology are the obvious reasons that separate research communities today exist in formal methods, fault-tolerance, and cyber-security. But there are also technical reasons for these communities to have remained independent despite their shared goal of facilitating the construction of systems we can depend on. The different concerns being addressed by the communities justify different and (surprisingly) incompatible foundations. With nations, organizations, and individuals growing ever more dependent on computing systems, the time is ripe to revisit and repudiate the technical rationale for these incompatibilities.

Acknowledgments. Brian Randall is not only the impetus for this paper (with this volume marking his 75^{th} birthday) but his research is also a foundation and inspiration. He was an early—and remains a most vocal—proponent that cyber-security and fault-tolerance would be better served by a single research community devoted to the broader agenda he and others term “dependability.”

References

1. Alpern, B., Schneider, F.B.: Defining liveness. *Information Processing Letters* 21(4), 181–185 (1985)
2. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. *Distributed Computing* 2(3), 117–126 (1987)
3. Clarkson, M., Schneider, F.B.: Hyperproperties. *Journal of Computer Security* 18(6), 1157–1210 (2010)
4. Forrest, S., Somayaji, A., Ackley, D.H.: Building diverse computer systems. In: *Proc. 6th Workshop on Hot Topics in Operating Systems*, pp. 67–72. IEEE Computer Science Press, Los Alamitos (1997)
5. Knight, J.C., Leveson, N.G.: An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering* 12(1), 96–109 (1986)
6. Lamport, L.: Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering* 3(2), 125–143 (1977)
7. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. *ACM Transactions on Programming Languages* 4(3), 382–401 (1982)
8. Randell, B.: On failures and faults. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 18–39. Springer, Heidelberg (2003)
9. Roeder, T., Schneider, F.B.: Proactive obfuscation. *ACM Transactions on Computing Systems* 28(2) (2010)
10. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22(4), 299–319 (1990)
11. Xu, J., Kalbarczyk, Z., Iyer, R.K.: Transparent runtime randomization for security. In: *Proc. 22nd International Symposium on Reliable Distributed Systems*, pp. 260–269. IEEE Computer Science Press, Los Alamitos (2003)