Blueprint for a Science of Cybersecurity*

Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

May 24, 2011

1 Introduction

A secure system must defend against all possible attacks—including those unknown to the defender. But defenders, having limited resources, typically develop defenses only for attacks they know about. New kinds of attacks are then likely to succeed. So our growing dependence on networked computing systems puts at risk individuals, commercial enterprises, the public sector, and our military.

The obvious alternative is to build systems whose security follows from first principles. Unfortunately, we know little about those principles. We need a *science of cybersecurity* (see Box 1) that puts the construction of secure systems onto a firm foundation by giving developers a body of laws for predicting the consequences of design and implementation choices. The laws should

- transcend specific technologies and attacks, yet still are applicable in real settings,
- introduce new models and abstractions, thereby bringing pedagogical value besides predictive power, and

*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.

What is a Science? (Box 1)

The term *science* has evolved in meaning since Aristotle used it to describe a body of knowledge. To many, it connotes knowledge obtained by systematic experimentation, so they take that process as the defining characteristic of a science. The natural sciences satisfy this definition.

Experimentation helps in forming and then affirming theories or laws that are intended to offer verifiable predictions about man-made and natural phenomena. It is but a small step from science as experimentation to science as laws that accurately predict phenomena. The status of the natural sciences remains unaffected by changing the definition of a science in this way. But computer science now joins. It is the study of what processes can be automated efficiently; laws about specification (problems) and implementations (algorithms) are a comfortable way to encapsulate such knowledge.

- facilitate discovery of new defenses as well as describing non-obvious connections between attacks, defenses, and policies, thus providing a better understanding of the landscape.

The research needed to develop this science of cybersecurity must go beyond the search for vulnerabilities in deployed systems and beyond the development of defenses for specific attacks. Yet, use of a science of cybersecurity when implementing a system should not be equated with implementing absolute security or even with concluding that security requires perfection in design and implementation. Rather, a science of cybersecurity would provide—independent of specific systems—a principled account for techniques that work, including assumptions they require and ways one set of assumptions can be transformed or discharged by another. It would articulate and organize a set of abstractions, principles, and trade-offs for building secure systems, given the realities of the threats and of our cybersecurity needs.

The field of cryptography comes close to exemplifying the kind of science base we seek. The focus in cryptography is on understanding the design and limitations of algorithms and protocols to compute certain kinds of results (for example, confidential or tamperproof or attributed) in the presence of certain kinds of adversaries who have access to some, but not all, information involved in the computation. Cryptography, however, is but one of

many cybersecurity building blocks. A science of cybersecurity would have to encompass richer kinds of specifications, computing environments, and adversaries. Peter Neumann [11] summarized the situation well when he opined about implementing cybersecurity "If you think cryptography is the answer to your problem, then you don't know what your problem is.".

An analogy with medicine can be instructive for contemplating benefits we might expect from a science of cybersecurity. Some health problems are best handled in a reactive manner. We know what to do when somebody breaks a finger, and each year we create a new influenza vaccine in anticipation of the flu season to come. But only after making significant investments in basic medical sciences are we starting to understand the mechanisms by which cancers grow, and a cure seems to require that kind of deep understanding. Moreover, nobody believes disease will some day be a "solved problem". We make enormous strides in medical research, yet new threats emerge and old defenses (for example, antibiotics) lose their effectiveness. Like good health, cybersecurity is never going to be a "solved problem." Attacks co-evolve with defenses and in ways to disrupt each new task that is entrusted to our networked systems. As with medical problems, some attacks are best addressed in a reactive way, while others are not. But our success in developing all defenses will benefit considerably from having laws that constitute a science of cybersecurity.

This paper gives one perspective on the shape of that science and its laws. Subjects that might be characterized in laws are discussed in section 2. Then, section 3 illustrates by giving concrete examples of laws. The relationship that a science of cybersecurity would have with existing branches of computer science is explored in section 4.

2 Laws about What?

In the natural sciences, quantities found in nature are related by laws: $E = mc^2$, $PV = nRT$, etc. Continuous mathematics is used to specify these laws. Continuous mathematics, however, is not intrinsic to the notion of a scientific law—predictive power is. Indeed, laws that govern digital computations are often most conveniently expressed using discrete mathematics and logical formulas. Laws for a science of cybersecurity are likely to follow suit, because these, too, concern digital computation.

But what should be the subject matter of these laws? To be deemed *secure*, a system should, despite *attacks*, satisfy some prescribed *policy* that specifies what the system must do (for example, deliver service) and what

it must not do (for example, leak secrets). And *defenses* are the means we employ to prevent a system from being compromised by attacks. This account suggests we strive to develop laws that relate attacks, defenses, and policies.

For generality, we should prefer laws that relate classes of attacks, classes of defenses, and classes of policies, where the classification exposes essential characteristics. Then we can look forward to having laws like "Defenses in class $\mathcal{D}$ enforce policies in class $\mathcal{P}$ despite attacks from class $\mathcal{A}$" or "By composing defenses from class $\mathcal{D}'$ and class $\mathcal{D}''$ a defense is constructed that resists the same attacks as defenses from class $\mathcal{D}$." Appropriate classes are crucial in order for a science of cybersecurity to be relevant.

2.1 Classes of Attacks

A system's *interfaces* define the sole means by which an environment can change or sense the effects of system execution. Some interfaces have clear embodiment to hardware: the keyboard and mouse for inputs, a graphic display or printer for outputs, and a network channel for both inputs and outputs. Other hardware interfaces and methods of input/output will be less apparent, and some are quite obscure. For example, Halderman et al. [7] show how lowering the operating temperature of a memory board facilitates capture of secret cryptographic keys through what they term a *cold boot* attack. The temperature of the environment is, in effect, an input to a generally overlooked hardware interface. Most familiar are interfaces created by software. The operating system interface often provides ways for programs to communicate overtly through system calls and shared memory or covertly through various side channels (such as battery level or execution timings).

Since (by definition) interfaces provide the only means for influencing and sensing system execution, interfaces necessarily constitute the sole avenues for conducting attacks against a system. The set of interfaces and the specific operations involved is thus one obvious basis for defining classes of attacks. For example, we might distinguish attacks (such as SQL-injections) that exploit overly-powerful interfaces from attacks (such as buffer overflows) that exploit insufficiently conservative implementations. Another basis for defining classes of attacks is to characterize the information or effort required for conducting the attack. With some cryptosystems, for instance, efficient techniques exist for discovering a decryption key if samples of ciphertext with corresponding plaintext are available for that key, but these techniques do not work when only ciphertext is available.

A given input might cause some policies to be violated but not others. So whether an input constitutes an attack on a given system could depend on the policy that system is expected to enforce. This dependence suggests that classes of attacks could be defined in terms of what policies they compromise. The definition of denial-of-service attacks, for instance, equates a class of attacks with system availability policies.

For attacks on communications channels, cryptographers introduce classifications based on the computational power or information available to the attacker. For example, *Dolev-Yao attackers* are limited to reading, sending, deleting, or modifying fields in messages being sent as part of some protocol execution [5]. (The altered traffic confuses the protocol participants, and they unwittingly undertake some action the attacker desires.) But it is not obvious how to generalize these attack classes to systems that implement more complex semantics than message delivery and that provide operations beyond reading, sending, deleting, or modifying messages.

Finally, the role of people in a system can be a basis for defining classes of attacks. Security mechanisms that are inconvenient will be ignored or circumvented by users; security mechanisms that are difficult to understand will be misused (with vulnerabilities introduced as a result). Distinct classes of attacks can thus be classified according to how or when the human user is fooled into empowering an adversary. Phishing attacks, which enable theft of passwords and ultimately facilitate identity theft, are one such class of attacks.

2.2 Classes of Policies

Traditionally, the cybersecurity community has formulated policies in terms of three kinds of requirements.

> **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.

This classification, as it now stands, is likely to be problematic as a basis for the laws that form a science of cybersecurity.

One problem is the lack of widespread agreement on mathematical definitions for confidentiality, integrity, and availability. A second problem is

Trace Properties, Safety, and Liveness (Box 2)

A *specification* for a sequential program would characterize for each input whether the program terminates and what outputs it produces. This characterization of execution as a relation is inadequate for concurrent programs. Lamport [9] introduced *safety* and *liveness* to describe the more expressive class of specifications that are needed for this setting. Safety asserts that no "bad thing" happens during execution and liveness asserts that some "good thing" happens.

A *trace* is a (possibly infinite) sequence of states; a *trace property* is a set of traces, where each trace in isolation satisfies some characteristic predicate associated with that trace property. Examples include *partial correctness* (the first state satisfies the input specification and any terminal state satisfies the output specification) and *mutual exclusion* (in each state, the program for at most one process designates an instruction in a critical section). Not all sets of traces define trace properties. *Information flow*, which stipulates a correlation between the values of the two variables across all traces, is an example. This set of traces does not have a characteristic predicate that depends only on each individual trace, so the set is not a trace property.

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.

that the three kinds of requirements are not orthogonal. For example, secret data can be protected simply by corrupting it so that the resulting value no longer accurately conveys the true secret value, thus trading integrity for confidentiality.[1] As a second example, any confidentiality property can be satisfied by enforcing a weak enough availability property, because a system that does nothing cannot be accessed by attackers to learn secret information.

[1] Clarkson and Schneider [4] use information theory to derive a law that characterizes the trade-off between confidentiality and integrity for database-privacy mechanisms.

Contrast this state of affairs with trace properties, where safety ("no 'bad thing' happens") and liveness ("some 'good thing' happens") are orthogonal classes.[2] Moreover, there is added value when requirements are formulated in terms of safety and liveness, because safety and liveness are each connected to a proof method. Trace properties, though, are not expressive enough for specifying all confidentiality and integrity policies. The class of hyperproperties [3], a generalization of trace properties, is. And hyperproperties include safety and liveness classes that enjoy the same kind of orthogonal decomposition that exists for trace properties. So hyperproperties are a promising candidate for use in a science of cybersecurity.

Any classification of policies is likely to be associated with some kind of system model and, in particular, with the interfaces the model defines (hence the operations available to adversaries). For example, we might model a system in terms of the set of possible indivisble state transitions that it performs while operating, or we might model a system as a black box that reads information streams from some channels and outputs on others. Sets of indivisible state transitions are a useful model for expressing laws about classes of policies enforced by various OS mechanisms (for example, reference monitors versus code rewriting) which themselves are concerned with allowed and disallowed changes to system state; stream models are often used for quantifying information leakage or corruption in output streams. We should expect that a science of cybersecurity will not be built around a single model or around a single classification of policies.

2.3 Classes of Defenses

A large and varied collection of different defenses can be found in the cybersecurity literature.

Program analysis and rewriting form one natural class characterized by expending the effort for deploying the defense (mostly) prior to execution. This class of defenses, called *language-based security*, can be further subdivided according to whether rewriting occurs (it might not occur with typechecking, for example) and according to the work required by the analysis and/or the rewriting. The undecidability of certain analysis questions and the high computation costs of answering others is sometimes a basis for further distinguishing *conservative* defenses—those analysis methods that can reject as being insecure programs that actually are secure, and those rewriting methods that add unnecessary checks.

[2]Formal definitions of trace properties, safety, and liveness are given in Box 2 for those readers who are interested.

Run-time defenses have, as their foundation, only a few basic mechanisms.

> **Isolation.** Execution of one program is somehow prevented from accessing interfaces that are associated with the execution of others. Examples include physically isolated hardware, virtual machines, and processes (which, by definition, have isolated memory segments).
>
> **Monitoring.** A *reference monitor* is guaranteed to receive control whenever any operation in some specified set is invoked; it further has the capacity to block subsequent execution, which it does to prevent an operation from proceeding when that execution would not comply with whatever policy is being enforced. Examples include memory mapping hardware, processors having modes that disable certain instructions, OS kernels, and firewalls,
>
> **Obfuscation.** Code or data is transmitted or stored in a form that can be understood only with knowledge of a secret. That secret is kept from the attacker, who then is unable to abuse, understand, or alter in a meaningful way the content being protected. Examples include data encryption, digital signatures, and program transformations that increase the work factor needed to craft attacks.

Obviously, a classification of run-time defenses could be derived from this taxonomy of mechanisms.

Another way to view defenses is in terms of trust relocation. For example, by running an application under control of a reference monitor we relocate trust in that application to trust in the reference monitor. This trust-relocation view of defenses invites discovery of general laws that govern how trust in one component can be replaced by trust in another.

We know that it is always possible for trust in an analyzer to be relocated to a proof checker—simply have an analyzer that concludes P also generate a proof of P. Moreover, this specific means of trust relocation is attractive, because proof checkers can be simple, hence easy to trust, whereas analyzers can be quite large and complicated. This suggests a related question: Is it ever possible to add defenses and transform one system into another, where the latter requires weaker assumptions about components being trusted? Perhaps trust is analogous to entropy in thermodynamics—something that can be reversed only at some cost (where "cost" corresponds to the strength of the assumptions that must be made)? Such questions are fundamental to the design of secure systems, and today's designers have no theory to help with answers. A science of cybersecurity could provide that foundation.

3 Laws Already on the Books

Attacks co-evolve with defenses, so a system that yesterday was secure might no longer be secure tomorrow. You can then wonder whether yesterday's science of cybersecurity would be made irrelevant by new attacks and new defenses. This depends on the laws, but if the classes of attacks, defenses, and policies are wisely constructed and sufficiently general then laws about them should be both interesting and long-lived. Examples of extant laws can provide some confirmation, and two (developed by the author) are discussed below.

3.1 Law: Policies and Reference Monitors

A developer who contemplates building or modifying a system will have in mind some class of policies that must be enforced. Laws that characterize what policies are enforced by given classes of defenses would be helpful here. Such laws have been derived for various defenses. Below, we discuss a law [13] concerning reference monitors.

The policy enforced by a reference monitor is the set of traces that correspond to executions in which the reference monitor does not block any operation. This set is a trace property, because whether the reference monitor blocks an operation in a trace depends only on the contents of that trace (specifically, the preceding operations in that trace). Moreover, this trace property is safety; the set of finite sequences that end in an operation the reference monitor blocks constitutes the "bad thing". We conclude:

Law. All reference monitors enforce trace properties that are safety.

This law, for example, implies that a reference monitor cannot enforce an information flow policy, since (as discussed in Box 2) information flow is not a trace property. However, the law does not preclude using a reference monitor to enforce a policy that is stronger and, by being stronger, implies that the information flow policy also will hold. But a stronger policy will deem insecure some executions the information flow policy does not. So such a reference monitor would block some executions that would be allowed by a defense that exactly enforces information flow. The system designer is thus alerted to a trade-off—employing a reference monitor for information flow policies brings overly conservative enforcement.

The above law also suggests a new kind of run-time defense mechanism [6]. For every trace property Ψ that is safety, there exists an automaton m_Ψ that accepts the set of traces in Ψ [2]. Automaton m_Ψ is a reference

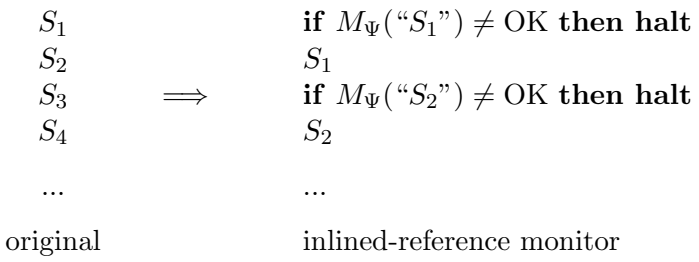

Figure 1: Inlined reference monitor example

monitor for Ψ because, by definition, it rejects traces that violate Ψ. So if code M_Ψ that simulates m_Ψ is invoked before every instruction in some given program S, then the result will be a new program that behaves just like S except it halts rather than executing an instruction that violates policy Ψ. This is depicted in Figure 1, where invocation $M_\Psi(x)$ simulates the transition that automaton m_Ψ makes for input symbol x and repeatedly returns OK until automaton m_Ψ would reject the sequence of inputs it has processed. Thus, the statement

$$\textbf{if} \quad M_\Psi(\text{“}S_i\text{”}) \neq \text{OK} \quad \textbf{then halt} \tag{1}$$

in Figure 1 immediately prior to a program statement S_i causes execution to terminate if next executing S_i would violate the policy defined by automaton m_Ψ—that is, if executing S_i would cause policy Ψ to be violated.

Such *inlined reference monitors* can be more efficient at run-time than traditional reference monitors, because a context switch is not required each time an inlined reference monitor is invoked. However, an inlined reference monitor must be installed separately in each program whose execution is being monitored, whereas a traditional reference monitor can be written and installed once and for all. The per-program installation does mean that inlined reference monitors can enforce different policies on different programs, an awkward functionality to support with a single traditional reference monitor. And per-program installation also means that code (1) inserted to simulate m_Ψ can be specialized and simplified, thereby allowing unnecessary checks to be eliminated for inlined reference monitors.

3.2 Law: Attacks and Obfuscators

We define a set of programs to be *diverse* if all implement the same functionality but differ in their implementation details. Diverse programs are less prone to having vulnerabilities in common, because attacks often depend on

memory layout and/or instruction sequence specifics. But building multiple distinct versions of a program is expensive.[3] So system implementors have turned to mechanical means for creating sets comprising diverse versions of a given program.

For mechanically-generated diversity to work as a defense, not only must implementations differ (so they have few vulnerabilities in common), but the differences must be kept secret from attackers. For example, buffer overflow attacks are generally written relative to some specific run-time stack layout. Alter this layout by rearranging the relative locations of variables as well as the return address on the stack, and an input designed to perpetrate an attack for the original stack layout is unlikely to succeed. But if the new stack layout were known by the adversary, then crafting an attack again becomes straightforward.

Programs to accomplish such transformations have been called *obfuscators*. An obfuscator τ takes two inputs—a program S and a secret key K—and produces a *morph*, which is a program $\tau(S, K)$ whose semantics is equivalent to S but whose implementation differs from S and from morphs generated with other keys. K specifies which exact transformations are applied in producing morph $\tau(S, K)$. Note that since S and τ are assumed to be publicly known, knowledge of K would enable an attacker to learn implementation details for successfully attacking morph $\tau(S, K)$.

Different classes of transformations are more or less effective in defending against the various different classes of attacks. This correspondence is important when designing a set of defenses for a given threat model, but knowing the specific correspondences is not the same as knowing the overall power of mechanically-generated diversity as a defense. That defensive power for programs written in a C-like language has been partially characterized in a set of laws [12]. Each *Obfuscator Law* establishes, for a specific (common) type system T_i and obfuscator τ_i pair, what is the relationship between two sets of attacks—those blocked when type system T_i is enforced versus those that cause execution of a morph $\tau_i(S, K)$ to abort for some secret key K.

The Obfuscator Laws do not completely quantify the difference between the effectiveness of type checking and obfuscation. But the laws are noteworthy for a science of cybersecurity because they circumvent the difficult problem of reasoning about attacks not yet invented. Laws about classes of known attacks risk irrelevance as new attacks are discovered. By formu-

[3]There is also experimental evidence [8] that distinct versions built by independent teams nevertheless share vulnerabilities.

lating the Obfuscator Laws in terms of a relation between sets of attacks, the need to identify or enumerate individual attacks is avoided. To wit, the class of attacks that type-checking defends against is not known and not given, yet the power of obfuscation to defend against an attack can now be meaningfully conveyed relative to the power of type-checking.

4 The Science In Context

A science of cybersecurity would build on knowledge from several existing areas of computer science. The connections to formal methods, fault-tolerance, and experimental computer science are nuanced; they are discussed below. However, cryptography, information theory, and game theory are also likely to be valuable sources of abstractions and laws. Finally, the physical sciences surely have a role to play—not only in matters of physical security but also for understanding unconventional interfaces to real devices that attackers might exploit (as exemplified by the cold boot attacks mentioned in section 2.1).

Formal Methods. Attacks are possible only because a system we deploy has flaws in its implementation, design, specification, or requirements. Eliminate the flaws and we eliminate the need to deploy defenses. But even when the systems on which we rely aren't being attacked, we should want confidence that they will function correctly. The presence of flaws undermines that confidence. So cybersecurity is not the only compelling reason to eliminate flaws.

The focus of formal methods research is on methods for gaining confidence in a system by using rigorous reasoning, including programming logics and model checkers.[4] This work has been remarkably successful with small systems or small specifications. It is used by companies like Microsoft to validate device drivers and Intel to validate chip designs. It is also the engine behind strong type-checking in modern programming languages (for example, Java and C#) and various code-analysis tools used in security audits.

Further developments in formal methods could serve a science of cybersecurity well. However, to date, work in formal methods has been based on trace properties or something with equivalent expressive power. This foundation allows mathematically elegant characterizations for whether a

[4]Other areas of software engineering are concerned with gaining confidence in a system through the use of experimentation (for example, testing) or management (for example, strictures on development processes).

Satisfies and Refinement (Box 3)

A program S can be modeled as a trace property Σ_S containing all sequences of states that could arise from executing S, and a specific execution of S satisfies a trace property P if the trace modeling that execution is in P. Thus, S *satisfies* P if and only if $\Sigma_S \subseteq P$ holds.

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 "satisfies" is closed under refinement. If S' refines S and S satisfies P, then S' satisfies P. 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 can be constructed by *step-wise refinement*.

With richer classes of policies, "satisfies" is unfortunately not 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^* enforces the confidentiality policy that no information flows between x and y, whereas (refinement) $S_{x=y}$ does not. Satisfies for the confidentiality policy is not closed under refinement, and step-wise refinement is not sound for deriving programs that satisfy this policy.

program satisfies a specification and for justifying step-wise refinement of programs. But trace properties are not adequately expressive for specifying all confidentiality, integrity, and availability policies, and stepwise refinement is not sound for these richer policies. (A mathematical justification of this limitation is provided in Box 3 for the interested reader.) So the foundations of today's formal methods would have to be changed to something with the expressiveness of hyperproperties—no small feat.

Byzantine Fault-Tolerance. A system is considered *fault-tolerant* if it will continue operating correctly even though some of its components exhibit

faulty behavior. Fault-tolerance is usually defined relative to a *fault model* that defines assumptions about what components can become faulty and what kinds of behaviors faulty components might exhibit. In the *Byzantine* fault model [10], faulty components are permitted to collude and to perform arbitrary state transitions. 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 for the Byzantine fault model, we ensure that the system can tolerate all behaviors that in practice could be exhibited by its faulty components.

The basic recipe for implementing such *Byzantine fault-tolerance* is well understood. We assume that the output of every component is a function of the preceding sequence of inputs. Each component that might fail is replaced by $2t+1$ replicas, where these replicas all receive the same sequence of inputs. Provided that t or fewer replicas are faulty, then the majority of the $2t + 1$ will be correct. These correct replicas will generate identical correct outputs, so the majority output from all replicas is unaffected by the behaviors of faulty components.

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 security by implementing Byzantine fault-tolerance. Unfortunately, the argument oversimplifies, and the conclusion is unsound.

- Replication, if anything, creates more opportunities for attackers to learn confidential information. So enforcement of confidentiality is not improved by the replication required for implementing Byzantine fault-tolerance. And storing encrypted data—even when a different key is used for each replica—does not solve the problem, if replicas actually must themselves be able to decrypt and process the data they store.

- Physically-separated components connected only by narrow-bandwidth channels are generally observed to exhibit uncorrelated failures. But physically separated replicas still will share many of the same vulnerabilities (because they will use the same code) and, therefore, will 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.

What should be clear, though, is that mechanically-generated diversity creates a kind of independence that can be a bridge from Byzantine fault-tolerance to attack tolerance. The Obfuscation Laws discussed in section 3.2 are a first step in this direction.

Experimental Computer Science. The code for a typical operating system can fit on a disk, and all of the protocols and interconnection that comprise the Internet are known. Yet the most efficient way to understand the emergent behavior of the Internet is not to study the documentation and program code—it is to apply stimuli and make measurements in a controlled way. Computer systems are frequently too complex to admit predictions about their behaviors. So just as experimentation is useful in the natural sciences, we should expect to find experimentation an integral part of computer science.

Even though we might prefer to derive our cybersecurity laws by logical deduction from axioms, the validity of those axioms will not always be self-evident. We often will work with axioms that embody approximations or describe models, as is done in the natural sciences. (Newton's laws of motion, for example, ignore friction and relativistic effects.) Experimentation is the way to gain confidence in the accuracy of our approximations and models. And just as experimentation in the natural science is supported by laboratories, experimentation for a science of cybersecurity will require testbeds where controlled experiments can be run.

Experimentation in computer science is somewhat distinct from what is called "experimental computer science" though. Computer scientists validate their ideas about new (hardware or software) system designs by building prototypes. This activity establishes that hidden assumptions about reality are not being overlooked. Performance measurements then demonstrate feasibility and scalability, which are otherwise difficult to predict. And for artifacts that will be used by people (for example, programming languages and systems), a prototype may be the only way to learn whether key functionality is missing and what novel functionality is useful.

Since a science of cybersecurity should lead to new ideas about how to build systems and defenses, the validation of those proposals could require building prototypes. This activity is not the same as engineering a secure system. Prototypes are built in support of a science of cybersecurity expressly to allow validation of assumptions and observation of emergent behaviors. So, a science of cybersecurity will involve some amount of experimental computer science as well as some amount of experimentation.

5 Concluding Remarks

The development of a science of cybersecurity could take decades. The sooner we get started, the sooner we will have the basis for a principled set of solutions to the cybersecurity challenge before us. Recent new federal funding initiatives in this direction are a key step. It's now time for the research community to engage.

Acknowledgments

An opportunity to deliver the keynote at an NSF/NSA/IARPA workshop on Science of Security in Fall 2008 was the impetus for me to start thinking about what shape a science of cybersecurity might take. The feedback from the participants at that workshop as well as discussions with the other speakers at a summer 2010 Jasons meeting on this subject was quite helpful. My colleagues in the NSF "TRUST" Science and Technology Center have been a valuable source of feedback, as have Michael Clarkson and Riccardo Pucella. I am grateful to Carl Landwehr, Brad Martin, Bob Meushaw, Greg Morrisett, and Pat Muoio for comments on an earlier draft of this paper.

About the Author

Fred B. Schneider joined the Cornell University faculty in 1978, where he is now the Samuel B. Eckert Professor of Computer Science. He also is the Chief Scientist of the NSF "TRUST" Science and Technology Center, and he has been Professor at Large at the University of Tromso since 1996.

A fellow of the AAAS, ACM, and IEEE, Schneider was granted a D.Sc. *honoris causa* by the University of Newcastle-upon-Tyne in 2003. He was awarded membership in Norges Tekniske Vitenskapsakademi (Norwegian Academy of Technological Sciences) in 2010 and the U.S. National Academy of Engineering in 2011. Schneider serves on the Computing Research Association's board of directors and is a council member of the Computing Community Consortium, which catalyzes research initiatives in the computer science. He is also a member of the Defense Science Board and the NIST Information Security and Privacy Advisory Board. A frequent consultant to industry, Schneider co-chairs Microsoft's TCAAB external advisory board on trustworthy computing.

References and Further Reading

[1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185 (1985).

[2] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126 (1987).

[3] Michael Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210 (2010).

[4] Michael Clarkson and Fred B. Schneider. Quantification of integrity. *Proceedings 23rd IEEE Computer Security Foundations Symposium* (Edinburgh, UK, July 2010), 28–43.

[5] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory* IT-29:2, 198–208 (1983).

[6] Ulfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. *Proceedings 2000 IEEE Symposium on Security and Privacy* (Oakland, California, May 2000), IEEE Computer Society, Los Alamito, California, 246–255.

[7] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. *Proceedings 2008 USENIX Security Symposium* (San Jose, CA, July 2008), USENIX Association, 45–60.

[8] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software. Engineering* 12(1):96–109 (Jan. 1986).

[9] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143 (1977).

[10] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages* 4(3):382–401 (July 1982).

[11] Peter G. Neumann, quoted in Gina Kolata. "The Key Vanishes: Scientist Outlines Unbreakable Code" *New York Times*, Feb. 20, 2001.

[12] Riccardo Pucella and Fred B. Schneider. Independence from obfuscation: A semantic framework for diversity. *Journal of Computer Security* 18(5):701–749 (2010).

[13] Fred Schneider. Enforceable security policies. *ACM Transactions on Information and System Security* 3(1):30–50 (2000).