THE Next Wave
The National Security Agency's review of emerging technologies

GUEST Editor's column

Robert Meushaw

The world's most extensive case of cyberespionage, including attacks on US government and UN computers, was reported at the 2011 Black Hat conference by security firm McAfee. Concluding five years of investigation, McAfee analysts were "surprised by the enormous diversity of the victim organizations and were taken aback by the audacity of the perpetrators." *Wired* magazine recently broke a story revealing that "a computer virus has infected the cockpits of America's Predator and Reaper drones, logging pilots' every keystroke as they remotely fly missions over Afghanistan and other war zones." These are but two examples of what have become almost routine reports of failures in system security. Increasingly, these problems directly affect us in important parts of our daily lives. And even more alarming is the rapid growth in the breadth and severity of these spectacular failures.

How are such widespread problems possible after decades of investment in computer security research and development? This question has gained the attention of increasing numbers of security professionals over the past several years. An emerging view is that these problems demonstrate that we do not yet have a good understanding of the fundamental science of security. Instead of fundamental science, most system security work has focused on developing ad hoc defense mechanisms and applying variations of the "attack and patch" strategy that emerged in the earliest days of computer security. Our national reliance on networked information systems demands that we approach security engineering with the same rigor that we expect in other engineering disciplines. We should expect designers of our digital infrastructure to have a well understood scientific foundation and advanced analytic tools comparable to those used in the production of other critical assets such as bridges, aircraft, power plants, and water purification systems.

The National Security Agency, the National Science Foundation (NSF), and the Intelligence Advanced Research Projects Activity jointly responded to this problem by sponsoring a workshop in November 2008 to consider whether a robust science of security was possible and to describe what it might look like. Academic and industry experts from a broad set of disciplines including security, economics, human factors, biology, and experimentation met with government researchers to help lay the groundwork for potential future initiatives. Since that meeting, a number of programs focused on security science have been initiated, along with an effort to help build a robust collaboration community.

This issue of *The Next Wave* is focused upon the important topic of security science. Included are articles from six of the experts who attended the 2008 workshop and have continued to work in the area of security science. Carl Landwehr from NSF provides a few historical examples of the relationship between engineering and science and shows how these examples might help us understand the evolution of cybersecurity. Adam Shostack from Microsoft provides another perspective on how science evolves and describes some steps he considers necessary to advance the development of cybersecurity science. Roy Maxion from Carnegie Mellon University (CMU) calls for greater scientific rigor in the way experimental methods are applied to cybersecurity. Dusko Pavlovic from Oxford University provides a unique and unexpected model for security to reason about what a security science might be. Anupam Datta from CMU and John Mitchell from Stanford University describe some of their joint work in one of the core problem areas for security—how to compose secure systems from smaller building blocks. Alessandro Chiesa from the Massachusetts Institute of Technology and Eran Tromer from Tel Aviv University describe a novel approach based upon probabilistically checkable proofs to achieve trusted computing on untrusted hardware. Their insights may lead to new strategies for dealing with a host of security problems that are currently considered intractable, including supply chain security.

The capstone article for this issue of *The Next Wave,* contributed by Fred Schneider of Cornell University, methodically constructs a "blueprint" for security science. Building on his keynote at the 2008 workshop, Schneider suggests that security science should describe features and

relationships with *predictive* value rather than create defenses *reactively* responding to attacks. Schneider's blueprint outlines the foundation for a security science comprising a body of laws that allow meaningful predictions about system security.

Developing a robust security science will undoubtedly require a long-term effort that is both broad based and collaborative. It will also demand resources well beyond those available to any single organization. But even with a generally acknowledged need for science, the temptation will be to continue fighting security fires with a patchwork of targeted, tactical activities. Good tactics can win a battle but good strategy wins the war. We need to create a better strategy for computer security research. As we continue to struggle with daily battles in cyberspace, we should not forget to pursue the fundamental science—the fundamental strategy—that will help to protect us in the future.

Technical Director emeritus
Trusted Systems Research, NSA

Contents

The Next Wave is published to disseminate technical advancements and research activities in telecommunications and information technologies. Mentions of company names or commercial products do not imply endorsement by the US Government.

To receive printed copies of *The Next Wave,* please use the Internet address below and provide a mailing address and the number of copies requested. For more information, please contact us:

National Security Agency
Attn: Kathleen Prewitt, Managing Editor
Suite 6541
Ft. George G. Meade, MD 20755–6541
301.688.9604 | TNW@tycho.ncsc.mil

Vol. 19 | No. 2 | 2012

Cybersecurity: From engineering to science

Carl E. Landwehr

Engineers design and build artifacts—bridges, sewers, cars, airplanes, circuits, software—for human purposes. In their quest for function and elegance, they draw on the knowledge of materials, forces, and relationships developed through scientific study, but frequently their pursuit drives them to use materials and methods that go beyond the available scientific basis. Before the underlying science is developed, engineers often invent rules of thumb and best practices that have proven useful, but may not always work. Drawing on historical examples from architecture and navigation, this article considers the progress of engineering and science in the domain of cybersecurity.

Over the past several years, public interest has increased in developing a *science of cybersecurity,* often shortened to *science of security* [1, 2]. In modern culture, and certainly in the world of research, science is seen as having positive value. Things scientific are preferred to things unscientific. A scientific foundation for developing artifacts is seen as a strength. If one invests in research and technology, one would like those investments to be scientifically based or at least to produce scientifically sound (typically meaning reproducible) results.

This yearning for a sound basis that one might use to secure computer and communication systems against a wide range of threats is hardly new. Lampson characterized access control mechanisms in operating systems in 1971, over 40 years ago [3]. Five years later Harrison, Ruzzo, and Ullman analyzed the power of those controls formally [4]. It was 1975 when Bell and LaPadula [5], and Walter, et al. [6], published their respective state-machine based models to specify precisely what was intended by "secure system." These efforts, preceded by the earlier Ware and Anderson reports [7, 8] and succeeded by numerous attempts to build security kernel-based systems on these foundations, aimed to put an end to a perpetual cycle of "penetrate and patch" exercises.

Beginning in the late 1960's, Djikstra and others developed the view of programs as mathematical objects that could and should be proven correct; that is, their outputs should be proven to bear specified relations to their inputs. Proving the correctness of algorithms was difficult enough; proving that programs written in languages with informally defined semantics implemented the algorithms correctly was clearly infeasible without automated help.

In the late 1970's and early 1980's several research groups developed systems aimed at verifying properties of programs. Proving security properties seemed less difficult and therefore more feasible than proving general correctness, and significant research funding flowed into these verification systems in hopes that they would enable sound systems to be built.

This turned out not to be so easy, for several

reasons. One reason is that capturing the meaning of *security* precisely is difficult in itself. In 1985, John McLean's System Z showed how a system might conform to the Bell-LaPadula model yet still lack the security properties its designers intended [9]. In the fall of 1986, Don Good, a developer of verification systems, wrote in an email circulated widely at the time: "I think the time has come for a full-scale redevelopment of the logical foundations of computer security . . ." Subsequent discussions led to a workshop devoted to Computer Security Foundations, inaugurated in 1988, that has met annually since then and led to the founding of *The Journal of Computer Security* a few years later.

All of this is not to say that the foundations for a science of cybersecurity are in place. They are not. But the idea of searching for them is also not new, and it's clear that establishing them is a long-term effort, not something that a sudden infusion of funding is likely to achieve in a short time.

But lack of scientific foundations does not necessarily mean that practical improvements in the state of the art cannot be made. Consider two examples from centuries past:

The Duomo, the Cathedral of Santa Maria Del Fiore, is one of the glories of Florence. At the time the first stone of its foundations was laid in 1294, the birth of Galileo was almost 300 years in the future, and of Newton, 350 years. The science of mechanics did not really exist. Scale models were built and used to guide the cathedral's construction but, at the time the construction began, no one knew how to build a dome of the planned size. Ross King tells the fascinating story of the competition to build the dome, which still stands atop the cathedral more than 500 years after its completion, and of the many innovations embodied both in its design and in the methods used to build it [10]. It is a story of human innovation and what might today be called engineering design, but not one of establishing scientific understanding of architectural principles.

About 200 years later, with the advent of global shipping routes, the problem of determining the East-West position (longitude) of ships had become such an urgent problem that the British Parliament authorized a prize of £20,000 for its solution. It was expected that the solution would come from developments in mathematics and astronomy, and so the Board of Longitude, set up to administer the prize competition, drew heavily on mathematicians and astronomers. In fact, as Dava Sobel engagingly relates, the problem was solved by the development, principally by a single self-taught clockmaker named John Harrison, of mechanical clocks that could keep consistent time even in the challenging shipboard environments of the day [11].

FIGURE 1. The Duomo, the Cathedral of Santa Maria Del Fiore, is a story of human innovation and what might today be called engineering design, but not one of establishing scientific understanding of architectural principles.

I draw two observations from of these vignettes in relation to the establishment of a science of cybersecurity. The first is that scientific foundations frequently follow, rather than precede, the development of practical, deployable solutions to particular problems. I

claim that most of the large scale software systems on which society today depends have been developed in a fashion that is closer to the construction of the Florence cathedral or Harrison's clocks than to the model of specification and proof espoused by Dijkstra and others. The Internet Engineering Task Force (IETF) motto asserting a belief in "rough consensus and running code" [12] reflects this fundamentally utilitarian approach. This observation is not intended as a criticism either of Dijkstra's approach or that of the IETF. One simply must realize that while the search for the right foundations proceeds, construction will continue.

Second, I would observe that the establishment of proper scientific foundations takes time. As noted earlier, Newton's law of gravitation followed Brunelleschi by centuries and could just as well be traced all the way back to the Greek philosophers. One should not expect that there will be sudden breakthroughs in developing a scientific foundation for cybersecurity, and one shouldn't expect that the quest for scientific foundations will have major near-term effects on the security of systems currently under construction.

What would a scientific foundation for cybersecurity look like? Science can come in several forms, and these may lead to different approaches to a science of cybersecurity [13]. Aristotelian science was one of definition and classification. Perhaps it represents the earliest stage of an observational science, and it is seen here both in attempts to provide a precise characterization of what security means [14] but also in the taxonomies of vulnerabilities and attacks that presently plague the cyberinfrastructure.

A Newtonian science might speak in terms of mass and forces, statics and dynamics. Models of computational cybersecurity based in automata theory and modeling access control and information flow might fall in this category, as well as more general theories of security properties and their composability, as in Clarkson and Schneider's recent work on hyperproperties [15]. A Darwinian science might reflect the pressures of competition, diversity, and selection. Such an orientation might draw on game theory and could model behaviors of populations of machines infected by viruses or participating in botnets, for example. A science drawing on the ideas of prospect theory and behavioral economics developed by Kahneman, Tversky, and others might be used to model risk perception and decision-making by organizations and individuals [16].

FIGURE 2. Scientific foundations frequently follow, rather than precede, the development of practical, deployable solutions to particular problems; for example, mechanical clocks were invented only after determining the longitude of ships had become such an urgent problem that the British Parliament authorized a £20,000 prize for its solution.

In conclusion, I would like to recall Herbert Simon's distinction of science from engineering in his landmark book, *Sciences of the Artificial* [17]:

> Historically and traditionally, it has been the task of the science disciplines to teach about natural things: how they are and how they work. It has been the task of the engineering schools to teach about artificial things: how to make artifacts that have desired properties and how to design.

From this perspective, Simon develops the idea that engineering schools should develop and teach a *science of design*. Despite the complexity of the artifacts humans have created, it is important to keep in mind that they are indeed artifacts. The community has the ability, if it has the will, to reshape them to better meet its needs. A science of cybersecurity should help people understand how to create artifacts that provide desired computational functions without being vulnerable to relatively trivial attacks and without imposing unacceptable constraints on users or on system performance.

About the author

Carl E. Landwehr is an independent consultant in cybersecurity research. Until recently, he was a senior research scientist for the Institute for Systems Research at the University of Maryland, College Park. He received his BS in engineering and applied science from Yale University and his PhD in computer and communication sciences from the University of Michigan. Following a 23-year research career at the Naval Research Laboratory, he has for the past decade developed and managed research programs at the National Science Foundation and the Advanced Research Development Activity/Defense Technology Office/ Intelligence Advanced Research Projects Activity. He is interested in all aspects of trustworthy computing. In December 2010, he completed a four-year term as editor in chief of *IEEE Security & Privacy Magazine.*

References

[1] Evans D. Workshop report. NSF/IARPA/NSA Workshop on the Science of Security; Nov 2008; Berkeley, CA. Available at: http://sos.cs.virginia.edu/report.pdf

[2] JASON Program Office. Science of cyber-security, 2010. McLean (VA): The Mitre Corporation. Report No.: JSR-10-102. Available at: http://www.fas.org/irp/agency/dod/jason/cyber.pdf

[3] Lampson BW. Protection. In: *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems;* Mar 1971; Princeton, NJ; p. 437–443. Reprinted in: Operating Systems Review. 1974;8(1):18–24. DOI: 10.1.1.137.1119

[4] Harrison MA, Ruzzo WL, Ullman JD. Protection in operating systems. *Communications of the ACM.* 1976;19(8):461–471. DOI: 10.1145/360303.360333

[5] Walter KG, Ogden WF, Gilligan JM, Schaeffer DD, Schaen SL, Shumway DG. Initial structured specifications for an uncompromisable computer security system, 1975. Hanscom Air Force Base, Bedford (MA): Deputy for Command and Management Systems, Electronic Systems Division (AFSC). Report No.: ESD-TR-75-82, NTIS AD-A022 490.

[6] Bell DE, La Padula L. Secure computer system: Unified exposition and multics interpretation, 1975. Hanscom Air Force Base, Bedford (MA): Deputy for Command and Management Systems, Electronic Systems Division (AFSC). Report No.: ESD-TR-75-306, DTIC AD-A023588. Available at: http://nob.cs.ucdavis.edu/history/papers/bell76.pdf

[7] Ware W. Security controls for computer systems: Report of Defense Science Board task force on computer security, 1970. Washington (DC): The Rand Coporation for the Office of the Director of Defense Research and Engineering. Report No.: R609-1. Available at: http://nob.cs.ucdavis.edu/history/papers/ware70.pdf

[8] Anderson JP. Computer security technology planning study, 1972. L.G. Hanscom Field, Bedford (MA): Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC). Report No.: ESD-TR-73-51, Vol. I, NTIS AD-758 206. Available at: http://nob.cs.ucdavis.edu/history/papers/ande72a.pdf

[9] McLean J. A comment on the 'Basic Security Theorem' of Bell and LaPadula. *Information Processing Letters.* 1985;20(2):6770. DOI: 10.1016/0020-0190(85)90065-1

[10] King R. *Brunelleschi's Dome: How a Renaissance Genius Reinvented Architecture.* New York (NY): Walker Publishing Company; 2000. ISBN 13: 978-0-802-71366-7

[11] Sobel D. *Longitude: The True Story of a Lone Genius Who Solved the Greatest Scientific Problem of His Time.* New York (NY): Walker Publishing Company; 1995. ISBN 10: 0-802-79967-1

[12] Hoffman P, Harris S. The Tao of IETF: A novice's guide to the Internet Engineering Task Force. Network Working Group, The Internet Society. RFC 4677, 2006. Available at: http://www.rfc-editor.org/rfc/rfc4677.txt

[13] Cybenko G. *Personal communication,* Spring, 2010. Note: I am indebted to George Cybenko for this observation and the subsequent four categories.

[14] Avizienis A, Laprie JC, Randell B, Landwehr C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing.* 2004;1(1):11–33. DOI: 10.1109/TDSC.2004.2

[15] Clarkson MR, Schneider FB. Hyperproperties. *Journal of Computer Security.* 2010;18(6):1157–1210. DOI: 10.3233/JCS-2009-0393

[16] Kahneman D, Tversky A. Prospect theory: An analysis of decision under risk. *Econometrica.* 1979;47(2):263–291. DOI: 10.2307/1914185

[17] Simon HA. *Sciences of the Artificial.* 3rd ed. Cambridge (MA): MIT Press; 1996. ISBN 13: 978-0-262-69191-8

The evolution of information security

Adam Shostack

Before Charles Darwin wrote his most famous works, *The Origin of Species* and *The Descent of Man,* he wrote a travelogue entitled *The Voyage of the Beagle.* In it he describes his voyages through South and Central America. On his journey, he took the opportunity to document the variety of life he saw and the environments in which it existed. Those observations gave Darwin the raw material from which he was able to formulate and refine his theory of evolution.

Evolution has been called the best idea anyone ever had. That's in part because of the explanatory power it brings to biology and in part because of how well it can help us learn in other fields. Information security is one field that can make use of the theory of evolution. In this short essay, I'd like to share some thoughts on how we can document the raw material that software and information technology professionals can use to better formulate and refine their ideas around security. I'll also share some thoughts on how information security might evolve under a variety of pressures. I'll argue that those who adopt ideas from science and use the scientific method will be more successful, and more likely to pass on their ideas, than those who do not.

1. The information security environment

Information security is a relatively new field. Some of the first people to undertake systematic analysis are still working in the field. Because the field and associated degree programs are fairly recent, many of those working in information security have backgrounds or degrees in other fields. What's more, those involved in information security often have a deep curiosity about the world, leading them to learn about even more fields. Thus, we have a tremendous diversity of backgrounds, knowledge, skills, and approaches from which the information security community can draw. Between a virtual explosion of niches in which new ideas can be brought to bear, and many different organizations to test those ideas, we ought to have a natural world of mutation, experimentation, and opportunities to learn. We should be living in a golden age of information security. Yet many security experts are depressed and demoralized. Debora Plunkett, head of the NSA's Information Assurance Directorate has stated, "There's no such thing as 'secure' anymore." To put a pessimistic face on it, risks are unmeasurable, we run on hamster wheels of pain, and budgets are slashed.

In the real world, evolution has presented us with unimaginably creative solutions to problems. In the natural world, different ways of addressing problems lead to different levels of success. Advantages accumulate and less effective ways of doing things disappear. Why is evolution not working for our security practices? What's different between the natural world and information security that inhibits us from evolving our security policies, practices, and programs?

2. Inhibitors to evolution

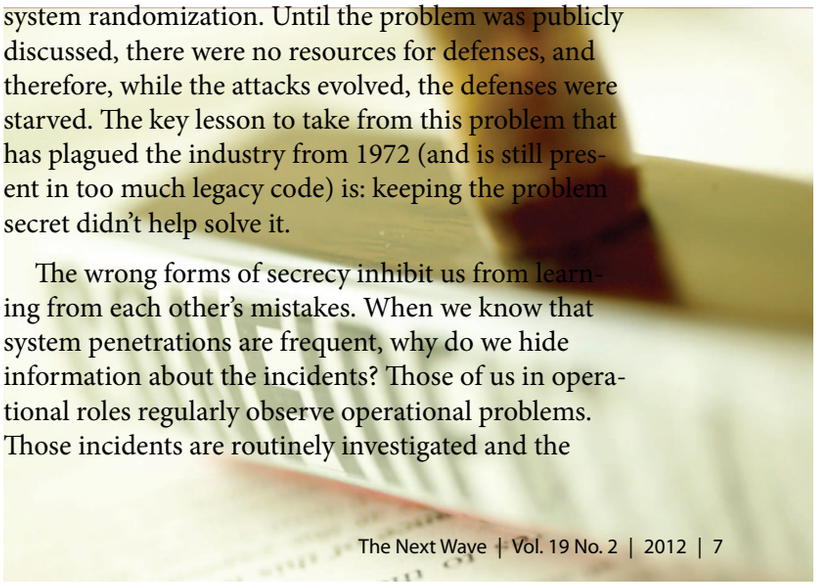
Information security programs are obviously not organisms that pass on their genes to new programs, and so discussions of how they evolve are metaphorical. I don't want to push the metaphor too far, but we ought to be able to do better than natural organisms because we can trade information without trading genes. Additionally, we have tremendous diversity, strong pressures to change, and even the advantage of being able to borrow ideas and lessons from each other. So why aren't we doing better?

Many challenges of building and operating effective security programs are well known. They include demonstrating business value, scoping, and demonstrating *why* something *didn't* happen. Let's focus on one reason that gets less attention: secrecy. To many who come to information security from a military background, the value of secrecy is obvious: the less an attacker knows, the greater the work and risk involved in an attack. It doesn't take a military background to see that putting a red flag on top of every mine makes a minefield a lot less effective. A minefield is effective precisely because it slows down attackers who have to expose themselves to danger to find a way through it. In information security operations, however, attacks can be made from a comfy chair on the other side of the world, with the attacker having first torn apart an exact copy of your defensive system in their lab. (This contrast was first pointed out by Peter Swire.)

We know that systems are regularly penetrated. Some say that all of them are. Despite that knowledge, we persist in telling each other that we're doing okay and are secure. Although the tremendously resilient infrastructures we've built work pretty well, we can and should do better.

For example, take the problem of stack smashing buffer overflows. The problem was clearly described in the public literature as early as 1972. According to Lance Hoffman, it was well known and influenced the design of the data flags in the main processors of the Burroughs B5500. The problem was passed down repeatedly through the 1980s and 1990s, and was exploited by the Morris Internet worm and many others. It was only after Aleph One published his paper "Smashing the stack for fun and profit" in 1996 that systematic defenses began to be created. Those defenses include StackGuard, safer string handling libraries, static analysis, and the useful secrecy in operating system randomization. Until the problem was publicly discussed, there were no resources for defenses, and therefore, while the attacks evolved, the defenses were starved. The key lesson to take from this problem that has plagued the industry from 1972 (and is still present in too much legacy code) is: keeping the problem secret didn't help solve it.

The wrong forms of secrecy inhibit us from learning from each other's mistakes. When we know that system penetrations are frequent, why do we hide information about the incidents? Those of us in operational roles regularly observe operational problems. Those incidents are routinely investigated and the



results of the investigation are almost always closely held. When we hide information about system failures, we prevent ourselves from studying those failures. We restrain our scientists from emulating Darwin's study of the variations and pressures that exist. We prevent the accumulation of data; we inhibit the development of observational methods; and we prevent scientific testing of ideas.

Let's consider what scientific testing of ideas means, and then get to a discussion of what ideas we might test.

3. Defining the problem

a. What is science?

For the sake of clarity, let me compare and contrast three approaches to problem solving and learning: science, engineering, and mathematics. Mathematics obviously underpins both science and engineering, but it will be helpful to untangle them a little.

At the heart of science is the falsification of hypotheses. Let me take a moment to explain what that means. A hypothesis is an idea with some predictive power. Examples include "everything falls at the same speed" (modulo friction from the air) and "gravity bends the path of light." Both of these hypotheses allow us to predict what will happen when we act. What's more, they're testable in a decisive way. If I can produce a material that falls faster than another in a vacuum, we would learn something fundamental about gravity. Contrast this with derivation by logic, where disproof requires a complex analysis of the proof. Science has many tools which center on falsifying hypotheses: the experiment, peer review, peer replication, publication, and a shared body of results. But at the heart of all science is the falsifiable hypothesis. Science consists of testable ideas that predict behavior under a range of circumstances, the welcoming of such tests and, at its best, the welcoming of the results. For more on the idea of falsifiability, I recommend Karl Popper's *Conjectures and Refutations*.

Science also overlaps heavily with engineering. Engineering concerns making tradeoffs between a set of constraints in a way that satisfies customers and stakeholders. Engineering can involve pushing boundaries of science, such as finding a way to produce lasers with shorter wavelengths, or pushing the limits of scientific knowledge. For example, when the original Tacoma Narrows Bridge finally buckled a little too hard, it drove new research into the aerodynamics of bridges.

The scientific approach of elimination of falsehood can be contrasted with mathematics, which constructs knowledge by logical proof. There are elements of computer security, most obviously cryptography, which rely heavily on mathematics. It does not devalue mathematics at all to note that interesting computer systems demonstrably have properties that are true but unprovable.

b. What is information security?

Information security is the assurance and reality that information systems can operate as intended in a hostile environment. We can and should usefully bring to bear techniques, lessons, and approaches from all sorts of places, but this article is about the intersection of science and security. So we can start by figuring out what sorts of things we might falsify. One easy target is the idea that you can construct a perfectly secure system. (Even what that means might be subject to endless debate, and not falsification.) Even some of the most secure systems ever developed may include flaws from certain perspectives. Readers may be able to think of examples from their own experience.

But there are other ideas that might be disproven. For example, the idea that computer systems with formal proofs of security will succeed in the marketplace can be falsified. It seems like a good idea, but in practice, such systems take an exceptionally long time to build, and the investment of resources in security proofs come at the expense of other features that buyers want more. In particular, it turns out that there are several probably false hypotheses about such computer systems:

- Proofs of security of design relate to the security of construction.
- Proofs of security of design or construction result in operational security.
- Proofs of security result in more secure systems than other security investments.
- Buyers value security above all else.

These are small examples but there are much larger opportunities to really study our activities and improve their outcomes for problems both technical and

human. As any practitioner knows, security is replete with failures, which we might use to test our ideas. Unfortunately, we rarely do so, opting instead for the cold comfort of approaches we know are likely to fail.

Why is it we choose approaches that often fail? Sometimes we don't know a better way. Other times, we feel pressure to make a decision that follows "standard practice." Yet other times, we are compelled by a policy or regulation that ignores the facts of a given case.

4. Putting it all together: A science of information security

So what ideas might we test? At the scale which the US government operates networks, almost any process can be framed as testable. Take "always keep your system up to date" or "never write down a password." Such ideas can be inserted into a sentence like "Organizations that dedicate *X* percent of their budget to practice *Y* suffer fewer incidents than those that dedicate it to practice *Z*."

Let me break down how we can frame this hypothesis:

1. The first choice I've made is to focus on organizations rather than individual systems. Individual systems are also interesting to study, but it may be easier to look to whole organizations.
2. The second choice is to focus on budget. Economics is always about the allocation of scarce resources. Money not spent on information security will be spent on other things, even if that's *just* returning it to shareholders or taxpayers. (As a taxpayer, I think that would be just fine.)
3. The third choice is to focus on outcomes. As I've said before, security is about outcomes, not about process (see http://newschoolsecurity.com/2009/04/security_is_about_outcome/). So rather than trying again to measure *compliance*, we look to incidents as a proxy for effectiveness. Of course, incidents are somewhat dependent on attacks being widely and evenly distributed. Fortunately, wide distribution of attacks is pretty much assured. Even distribution between various organizations is more challenging, but I'm confident that we'll learn to control for that over time.
4. The final choice is that of comparisons. We should compare our programs to those of other organizations, and to their choices of practices.

Of course, comparing one organization to another without consideration of how they differ might be a lot like comparing the outcomes of heart attacks in 40-year-olds to 80-year-olds. Good experimental design will require either that we carefully match up the organizations being compared or that we have a large set and are randomly distributing them between conditions. Which is preferable? I don't know, and I don't need to know today. Once we start evaluating outcomes and the choices that lead to them, we can see what sorts of experiments give us the most actionable information and refine them from there. We'll likely find several more testable hypotheses that are useful.

Each of the choices above can be reframed as a testable hypothesis of "does measuring this get us the results we want?" If you think the question of, "Do organizations that dedicate *X* percent of their budget to practice *Y* suffer fewer incidents than those that dedicate it to practice *Z*?" is interesting, then, before testing any ideas, bringing science to information security helps us ask more actionable questions.

Similarly, we can think about building outcome-oriented tests for technology. *Proof of concept* exploit code can be thought of as disproving the trivial hypothesis that, "This program has no exploitable vulnerability of class *X*." Since we know that programs usually have a variety of flaws associated with the languages used to construct them, we would expect many of those hypotheses to be false. Nevertheless, demonstration code can focus attention on a particular issue and help get it resolved. But we can aspire to more surprising hypotheses.

5. Next steps

Having laid out some of the challenges that face information security and some of what we will gain as we apply the scientific method, here is what we need to do to see those benefits:

1. **Robust information sharing (practices and outcomes).** We need to share information about what organizations are doing to protect their information and operations, and how those protections are working. Ideally, we will make this information widely available so that people of different backgrounds and skills can analyze it. Through robust and broad debate,

Robust information sharing

Robust hypothesis testing

Fast reaction and adaptation

we're more likely to overcome groupthink and inertia. Fortunately, the federal government already shares practice data in reports from the Office of the Inspector General and the Government Accountability Office. Outcome reporting is also available, in the form of data sent to the US Computer Emergency Readiness Team (US-CERT). The Department of Veterans Affairs publishes the information security reports it sends to Congress. Expanding on this information publication will accelerate our ability to do science.

2. **Robust hypothesis testing.** With the availability of data, we need to start testing some hypotheses. I suggest that nothing the information security community could do would make millions of people happier faster and at less risk than reducing password requirements. Testing to see if password complexity requirements have any impact on outcomes could allow many organizations to cut their help desk and password reset requirements at little cost to security.
3. **Fast reaction and adaptation.** Gunnar Peterson has pointed out that as technologies evolved from file transfer protocol (FTP) to hypertext transfer protocol (HTTP) to simple object access protocol (SOAP), security technologies have remained "firewalls and SSL." It can seem like the only static things in security are our small toolbox and our depression. We need to ensure that innovations by attackers are understood and called out in incident responses and that these innovations are matched by defenders in ways that work for each organization and its employees.

There are objections to these ideas of data sharing and testing. Let me take on two in particular.

The first objection is "This will help attackers." But information about defensive systems is easily discovered. For example, as the DEF CON 18 Social Engineering contest made irrefutable, calling employees on the phone pretending to be the help desk reveals all sorts of information about the organization. "Training and education" were clearly not effective for those organizations. If you think your training works well, please share the details, and perhaps someone will falsify your belief. My hypothesis is that every organization of more than a few hundred people has a great deal of information on their defenses which is easily discovered. (As if attackers need help anyway.)

The second objection is that we already have information-sharing agreements. While that is true, they generally don't share enough data or share the data widely enough to enable meaningful research.

Information security is held back by our lack of shared bodies of data or even observations. Without such collections available to a broad community of research, we will continue along today's path. That's not acceptable. Time after time, the scientific approach has demonstrated effectiveness at helping us solve thorny problems. It's time to bring it to information security. The first step is better and broader sharing of information. The second step is testing our ideas with that data. The third step will be to apply those ideas that have passed the tests, and give up on the superstitions which have dogged us. When we follow Darwin and

his naturalist colleagues in documenting the variety of things we see, we will be taking an important step out of the muck and helping information security evolve.

About the author

Adam Shostack is a principal program manager on the Microsoft Usable Security team in Trustworthy Computing. As part of ongoing research into classifying and quantifying how Windows machines get compromised, he recently led the drive to change Autorun functionality on pre-Win7 machines; the update has so far improved the protection of nearly 500 million machines from attack via universal serial bus (USB). Prior to Usable Security, he drove the *Security Development Lifecycle (SDL) Threat Modeling Tool* and *Elevation of Privilege: The Threat Modeling Game* as a member of the SDL core team. Before joining Microsoft, Adam was a leader of successful information security and privacy startups and helped found the Common Vulnerabilities and Exposures list, the Privacy Enhancing Technologies Symposium, and the International Financial Cryptography Association. He is coauthor of the widely acclaimed book, *The New School of Information Security.*

Further reading

Aleph One. 1996. Smashing the stack for fun and profit. *Phrack.* 1996;7(49). Available at: http://www.phrack.org/issues.html?issue=49&id=14#article

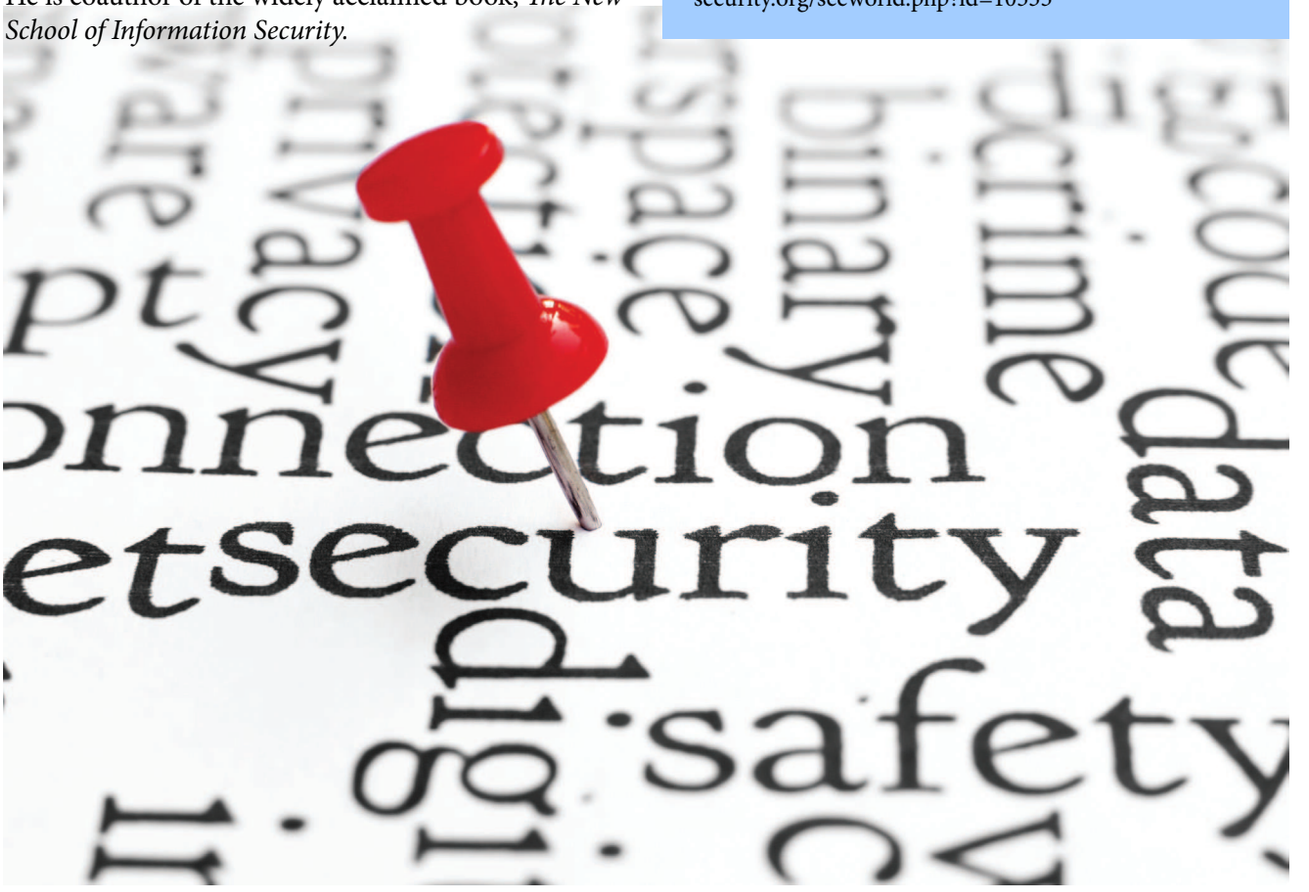
Anderson JP. Computer security technology planning study, 1972. L.G. Hanscom Field, Bedford (MA): Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC). Report No.: ESD-TR-73-51, Vol. I, NTIS AD-758 206. Available at: http://nob.cs.ucdavis.edu/history/papers/ande72a.pdf

Hoffman L. *Personal communication,* but see also the Burroughs tribute page available at: http://web.me.com/ianjoyner/Ian_Joyner/Burroughs.html

Popper K. *Conjectures and Refutations: The Growth of Scientific Knowledge.* London: Routledge; 1963. ISBN 13: 978-0-710-01966-0

Swire P. A model for when disclosure helps security: What is different about computer and network security? *Journal on Telecommunications and High Technology Law.* 2004;3(1):163–208.

Zorz Z. NSA considers its networks compromised. *Help Net Security.* 2010 Dec 17. Available at: http://www.net-security.org/secworld.php?id=10333



C.B. Jones and J.L. Lloyd (Eds.): Festschrift Randell, LNCS 6875, pp. 344–357, 2011.

Making experiments dependable |

Roy Maxion*

Abstract. In computer science and computer security we often do experiments to establish or compare the performance of one approach vs. another to some problem, such as intrusion detection or biometric authentication. An experiment is a test or an assay for determining the characteristics of the item under study, and hence experimentation involves measurements.

Measurements are susceptible to various kinds of error, any one of which could make an experimental outcome invalid and untrustworthy or undependable. This paper focuses on one kind of methodological error—confounding—that can render experimental outcomes inconclusive, but often without the investigator knowing it. Hence, valuable time and other resources can be expended for naught. We show examples from the domain of keystroke biometrics, explaining several different examples of methodological error, their consequences, and how to avoid them.

1. Science and experimentation

You wouldn't be surprised if, in a chemistry experiment, you were told that using dirty test tubes and beakers (perhaps contaminated with chemicals from a past procedure) could ruin your experiment, making your results invalid and untrustworthy. While we don't use test tubes in cyber security, the same admonition applies: keep your experiments clean, or the contamination will render them useless.

Keeping your glassware clean is part of the chem-lab methodology that helps make experimental measurements dependable, which is to say that the measurements have minimal error and no confounding variables. In cyber security we also need measurements that are dependable and error-free; undependable measurements make for undependable values and analyses, and for invalid conclusions. A rigorous experimental methodology will help ensure that measurements are valid, leading to outcomes in which we can have confidence.

A particularly insidious form of error is the confound—when the value of one variable or experimental phenomenon is confounded or influenced by the value of another. An example, as above, would be measuring the pH of a liquid placed in contaminated glassware where the influence of the contaminant on pH varied with the temperature of the liquid being measured. This is a confound, and to make things worse, the experimenter would likely be unaware of its presence or influence. The resulting pH values might be attributed to the liquid, to the temperature, or to the contaminant; they cannot be distinguished (without further experimentation). Similar measurement error can creep into cyber security experiments, making their measures similarly invalid.

This article describes some of the issues to be considered, and the rationales for decisions that need to be made, to ensure that an experiment is valid—that is, that outcomes can be attributed to only one cause (no alternative explanations for causal relations), and that experimental results will generalize beyond the experimental setting.

In the sections to follow, we first consider the hallmarks of a good experiment: repeatability, reproducibility and validity. Then we focus on what is arguably the most important of these—validity. We examine a range of threats to validity, using an experiment in

* The author is grateful for support under National Science Foundation grant number CNS-0716677. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author, and do not necessarily reflect the views of the National Science Foundation.

keystroke biometrics to provide examples. The experiment is laid out first, and is then critiqued; remedies for the violations are suggested. We close by suggesting simple ways to avoid the kinds of problems described here.

2. Hallmarks of a good experiment

There are clear differences between experiments that are well-designed and those that are not. While there may be many details that are different between the two, the main ones usually reduce to issues of repeatability (sometimes called reliability), reproducibility and validity. While our main focus here will be on validity, we will first look briefly at what each of the other terms means, just to put them all in context.

Repeatability refers to the variation in repeated measurements taken by a single person or instrument on the same item and under the same conditions; we seek high agreement, or consistency, from one measured instance to another [9]. That is, the experiment can be repeated in its entirety, and the results will be the same every time, within measurement error. For example, if you measure the length of a piece of string with a tape measure, you should get about the same result every time. If an experiment is not repeatable, even by the same person using the same measuring apparatus, then there is a risk that the measurement is wrong, and hence the outcome of the experiment may be wrong, too; but no one will realize it, and so erroneous values will be reported (and assumed to be correct by readers).

Reproducibility relates to the agreement of experimental results with independent researchers using similar but physically different test apparatus, and different laboratory locations, but trying to achieve the same outcome as was reported in a source article [9]. Measurements should yield the same results each time they are taken, irrespective of who does the measuring. Using the length-of-string example, if other people can measure that same piece of string in another setting using a similar measuring device, they should get about the same result as the first group did. If they don't, then the procedure is not reproducible; it can't be replicated. Reproduction (sometimes called replication) allows an assessment of the control on the operating conditions of the measurement procedure, i.e., the ability to reset the conditions to some desired state. Ultimately, replication reflects how well the procedure was operationalized.

FIGURE 1. Hallmarks of a good experiment.

Note that reproducibility doesn't mean hitting `return` and analyzing the same data set again with the same algorithm. It means conducting the entire experiment again, data collection and all. If an experiment is not reproducible, then it cannot be replicated by others in a reliable way. This means that no one will be able to verify that the experiment was done correctly in the first place, hence placing an air of untrustworthiness on the original results. Reproducibility hinges on operational definitions for the measures and procedures employed in the course of the experiment. An operational definition defines a variable or a concept in terms of the procedures or operations used to measure it. An operational definition is like a recipe or set of detailed instructions for describing or measuring something.

Validity relates to the logical well-groundedness of how the experiment is conducted, as well as the extent to which the results will generalize to circumstances beyond those in the laboratory. The next section expands on the concept of validity.

3. Validity

What does the term *valid* mean? Drawing from a standard dictionary, when some thing or some argument or some process is *valid,* it is well-grounded or justifiable; it is logically correct; it is sound and flawlessly reasoned, supported by an objective truth.

To conduct an experiment that was anything other than valid, in the above sense, would be foolish, and yet we see such experiments all the time in the literature. Sometimes we can see the flaws (which some would call *threats to validity*) directly in the experiment, and sometimes we can't tell, because authors do not report the details of how their experiments were conducted. Generally speaking, there are two kinds of validity—internal and external. Conceptually, these are pretty simple.

Internal validity. In most experiments we are trying to find out if A has a given effect on B, or if A causes B. To claim that A indeed causes B, the experiment must not offer any alternative causes nor alternative explanations for the outcome; if this is case, then the experiment is internally valid [8]. An alternative explanation for an experimental outcome can be due, for example, to confounded variables that have not been controlled.

For example, suppose we want to understand the cause of errors in programming. We recruit students in university programming classes (one class uses C, and the other uses Java). We ask all the students to write a program that calculates rocket trajectories. The results indicate that C programmers make more programming errors, and so we conclude that the C programming language is a factor in software errors. Drawing such a conclusion would be questionable, because there are other factors that could explain the results just as well. Suppose, for example, that the Java students were more advanced (juniors, not sophomores) than the C students. The outcome of the experiment could be due to the experience level of the students, just as much as it could be due to the language. Since we can't distinguish distinctly between experience level and language, we say that the experiment confounds two factors—language and experience—and is therefore not valid. Note that it can sometimes be quite difficult to ensure internal validity. Even if all the students are at the same experience level, if they self-selected Java vs C it would still allow for a confound in that a certain kind of student might be predisposed to select Java, and a different kind of student might be predisposed to select C. The two different kinds of students might be differentially good at one language or the other. The remedy for such an occurrence would be to assign the language-student pairs randomly.

External validity. In most experiments we hope that the findings will apply to all users, or all software, or all applications. We want the experimental findings to generalize from a laboratory or experimental setting to a much broader setting. To the extent that a study's findings generalize to a broader population (usually taken to be "the real world"), the experiment is externally valid [8]. If the findings are limited to the conditions surrounding the study (and not to broader settings), then the experiment lacks external validity. Another way to think about this is that external validity is the extent to which a causal relationship holds when there are variations in participants, settings and other variables that are different from the narrow ranges employed in the laboratory.

Referring back to our earlier example, suppose we were to claim that the experiment's outcome (that the C language promotes errors) generalizes to a set of programmers outside the experimental environment—say, in industry. The generalization might not hold, perhaps because the kind of problem presented to the student groups was not representative of the kinds of problems typically encountered in industry. This is an example of an experiment not generalizing beyond its experimental conditions to a set of conditions more general; it's not externally valid.

Trade-off between internal and external validity. It should be noted that not all experiments can be valid both internally and externally at the same time; it depends on the purpose of the experiment whether we seek high internal or high external validity. Typically there is a trade-off in which one kind of validity is sacrificed for the other. For example, laboratory experiments designed to answer a very focused question are often more internally valid than externally valid. Once a research question seems to have been settled (e.g., that poor exception handling is a major cause of software failure), then a move to a broader, more externally valid, experiment would be the right thing to do.

4. Example domain—keystroke biometrics

In this section we introduce the domain from which we draw concrete examples of experimental invalidities—keystroke biometrics.

Keystroke biometrics, or keystroke dynamics, is

the term given to the procedure of measuring and assessing a user's typing style, the characteristics of which are thought to be unique to a person's physiology, behavior, and habits. The idea has its origin in the observation that telegraph operators have distinctive patterns, called *fists,* of keying messages over telegraph lines. One notable aspect of fists is that they emerge naturally, as noted over a hundred years ago by Bryan & Harter, who showed that operators are distinctive due to the automatic and unconscious way their personalities express themselves, such that they could be identified on the basis of having telegraphed only a few words [1].

These measures of key presses and key releases, based largely on the timing latencies between keystrokes, are compared to a user profile as part of a classification procedure; a match or a non-match can be used to decide whether or not the user is authenticated, or whether or not the user is the true author of a typed sequence. For a brief survey of the keystroke literature, see [7].

We use keystroke dynamics as an example here for two reasons. First, it's easy to understand—much easier, for example, than domains like network protocols. If we're going to talk about flaws and invalidities in experiment design, then it's better to talk about an experiment that's easily understood; the lessons learned can be extended to almost any other domain and experiment. Second, keystroke dynamics shares many problems with other cyber-security disciplines, such as intrusion detection. Examples are classification and detection accuracy; selection of best classifier or detector; feature extraction; and concept drift, just to name a few. Again, problems solved in the keystroke domain are very likely to transfer to other domains where the same type of solution will be effective.

4.1. What is keystroke dynamics good for?

Keystroke dynamics is typically thought of as an example of the second factor in two-factor authentication. For example, for a user to authenticate, he'd have to know not only his own password (the first factor), but he would also have to type the password with a rhythm consistent with his own rhythm. An impostor, then, might know your password, but would not be able to replicate your rhythm, and so would not be allowed into the system. Another application, along a similar line, would be continuous re-authentication, in which the system continually checks to see that the typing rhythm matches that of the logged-in user, thereby preventing, say, insiders from masquerading as you. A third application would be what forensics experts call questioned-document analysis, which asks whether a particular user typed a particular document or parts of it. Finally, keystroke rhythms could be used to track terrorists from one cyber café to another, or to track a predator from one chat-room session to another.

4.2. How does keystroke dynamics work?

The essence of keystroke dynamics is that timing data are collected as a typist enters a password or other string. Each keystroke is timestamped twice; once on its downstroke and once on its upstroke. From those timings we can compute the amount of time that a key was held down (hold time) and the amount of time it took to transition from one key to the next (transition latency). The hold times and the latencies are called *features* of the typed password, and for a given typing instance these features would be grouped into a feature vector. For a 10-character password there would be eleven hold times and ten latencies if we include the `return` key.[a] If a typist enters a password many times, then the several resulting feature vectors can be assembled into a template which represents the central tendency of the several vectors. Each typist will have his or her own such template. These templates are formed during an enrollment period, during which legitimate users provide typing samples; these samples form the templates. Later, when a user wishes to log in, he types the password with the implicit claim that the legitimate user has typed the password. The keystroke dynamics system examines the feature vector of the presently-typed password, and classifies it as either belonging to the legitimate user or not. The classifier operates as an anomaly detector; if the rhythm of the typed password is a close enough match to the stored template, then the user is admitted to the system. The key aspect of this mechanism is the detector. In machine learning there are many such detectors, distinguished by the distance metrics that they use, such as Euclidean, Manhattan and Mahalanobis, among others [4]. Any of these detectors can be used in a keystroke

a. There are two kinds of latencies—keydown to keydown and keyup to keydown. Some researchers use one or the other of these, and some researchers use both. In our example we would have 31 features if we used both.

dynamics system; under some circumstances, some detectors work better than others, but it is an open research question as to which classifier is overall best.

5. A typical keystroke experiment

In this section we discuss several aspects of conducting a study in keystroke dynamics, we show what can go wrong, and we share some examples of how (in) validity can affect the outcome of a real experiment. We will discuss some examples and experimental flaws that are drawn from the current literature, although not all of the examples are drawn from a single paper.

Walkthrough. Let's walk through a typical experiment in keystroke dynamics, and we'll point out some errors that we've observed in the literature, why they're errors, how to correct them, and what the consequences might be if they're left uncorrected. Note that the objective of the experiment is to discriminate among users on the basis of their typing behavior, not on the basis of their typing behavior plus, possibly unspecified, other factors; the typing behavior needs to be isolated from other factors to make the experiment valid.

A typical keystroke dynamics experiment would test how well a particular algorithm can determine that a user, based on his typing rhythm, is or is not who he claims to be. In a keystroke biometric system, a user would present himself to the system with his login ID, thereby claiming to be the person associated with the ID. The system verifies this claim by two means: it checks that the password typed by the user is in fact the user's password; and it checks that the password is typed with the same rhythm with which the legitimate user would type it. If these two factors match the system's stored templates for the user, then the user is admitted to the system.

Checking that the correct password is offered is old hat; checking that its typing rhythm is correct is another matter. This is typically done by having the user "enroll" in the biometric component of the system. For different biometric systems the enrollment process is different, depending on the biometric being used; for example, if a fingerprint is used, then the user needs to present his fingerprint to the system so that the system can encrypt and store it for later matching against a user claiming to be that person who enrolled. For keystroke biometric systems, the process is similar; the user types his password several times so that the system can form a profile of the typing rhythm for later matching. The biometric system's detection algorithm is tested in two ways. In the first test, sample data from the enrolled user is presented to the system; the system should recognize that the user is legitimate. The second test determines whether the detector can recognize that an impostor is not the claimed user. This would be done by presenting the impostor's login keystroke sequence to the system, posing as a legitimate user. Across a group of legitimate users and impostors, the percentage of mistakes, or errors, serves as a gauge of how good the keystroke biometric system is. Several details concerning exactly how these tests are done can have enormous effects on the outcome. We turn now to those details.

What can go wrong? There are several parts of an experiment where things can go wrong. Most experiments measure something; the measuring apparatus can be flawed, producing flawed measurements. If the measurements are flawed, then the data will be flawed, and any analytical results and conclusions will be cast into doubt. The *way* that something is measured can be unsound; if you measure code complexity by counting the number of lines, you'll get a numerical outcome, but it may not be an accurate reflection of code complexity. The way or method of taking measurements is the biggest source of error in most experiments. Compounding that error is the lack of detail with which the measurement methodology is reported, often making it difficult to determine whether or not something went wrong. We turn now to specific examples of methodological problems.

Clock resolution and timing. Keystroke timings are based on operating-system calls to various timers. In the keystroke literature we see different timers being used by different researchers, with timing accuracies often reported to several decimal places. But it's not the accuracy (number of decimal places) of the timing that's of overriding importance; it's the resolution. When keystroke dynamics systems are written for Windows-based machines (e.g., Windows XP), it's usually the tick timer, or *Windows-event clock* [6] that's used; this has a resolution of 15.625 milliseconds (ms), corresponding to 64 updates per second. If done on a Unix system, the resolution is about 10 milliseconds. On some Windows systems the resolution can

be much finer if the QPC timer is used. The reason that timing resolution matters is not because people type as fast as one key every 15 milliseconds (66 keys per second); it's because the time *between* keystrokes can differ by less than 15 milliseconds. If some typists make key-to-key transitions faster than other ones, but the clock resolution is unable to separate them, then detection accuracy could suffer. One paper has reported a 4.2% change in error rate due to exactly this sort of thing [3]. A related issue is how you know what your clock resolution is. It's unwise to simply read this off the label; better to perform a calibration. A related paper explained how this is done in a keystroke dynamics environment [5]. A last word on timing issues concerns how the timestamping mechanism actually works; if it's subject to influence from the scheduler, then things like system load can change the accuracy of the timestamps.

The effect of clock resolution and timing is that they can interact with user rhythms as a confound. If different users type on different machines whose timing resolutions differ, then any distinctions made among users, based on timing, could be due to differences in user typing rhythms (timings) or they could be due to differences in clock resolutions. Moreover, since system load can influence keystroke timing, it's possible that rhythmic differences attributed to different users would be due to load differences, not to user differences. Hence we would not be able to claim distinctiveness based on user behavior, because this cannot be separated from timing errors induced by clock resolution and system load. If the purpose of the experiment is to differentiate amongst users on the basis of typing rhythm, then the confounds of clock resolution and system load must be removed. The simplest way to achieve this is to ensure that the experimental systems use the same clock, with the same resolution (as high as possible), and have the same operating load. This is possible in the laboratory by using a single system on which to collect data from all participants.

Keyboards. Experiments in keystroke dynamics require people to type, of course, and keyboards on which to do that typing. Most such experiments reported in the literature allow subjects to use whatever keyboard they want; after all, in the real world people do use whatever keyboard they prefer. Consequently, this approach has a lot of external validity. Unfortunately, the approach introduces a serious confound, too—a given keyboard, by its shape or character layout, is likely to influence a user's typing behavior. Different keyboards, such as standard, ergonomic, laptop, kinesis, natural, kinesis maxim split and so forth will shape typing in a way that's peculiar to the keyboard itself. In addition to the shape of the keyboard, the key pressures required to make electrical contact differ from one keyboard to another. The point is that not all keyboards are the same, with the consequence that users may type the same strings differently, depending on the keyboard and its layout. In the extreme, if everyone in the experiment used a different keyboard, you wouldn't be able to separate the effect of the keyboards from the effect of typing rhythm; whether your experimental results showed good separation of typists or not, you wouldn't know if the results were due to the typists' differences or to the differences among the keyboards. Hence you would not be able to conclude that typing rhythms differ among typists. This confound can be removed from the experiment by ensuring that all participants use the same (or perhaps same type of) keyboard. The goal of the experiment is to determine distinctiveness amongst typists based on their individual rhythms, not on the basis of the keyboards on which they type.

Stimulus items—what gets typed. Participants in keystroke biometrics experiments need to type something—the stimulus item in the experiment. While there are many kinds of stimuli that could be considered (e.g., passwords, phrases, paragraphs, transcriptions, free text, etc.), we focus on short, password-like strings. There are two fundamental issues: string contents and string length.

String contents. By contents we mean simply the characters contained in the password being typed. Two contrasting examples would be a strong password, characterized by containing shift and punctuation characters, as opposed to a weak password, characterized by a lack of the aforementioned special characters. It's easy to see that if some users type strong passwords, and other users type weak passwords, then any discrimination amongst users may not be solely attributable to differences among users; it may be attributable to intrinsic differences between strong and weak passwords that cause greater rhythmic variability in one or the other. The reason may be that strong passwords are hard to type, and weak ones aren't. So we may be discriminating not on the basis of user

rhythm, but on the basis of typing difficulty which, in turn, is influenced by string content. To eliminate this confound, the experimenter should not allow users to choose their own passwords; the password should be chosen by the experimenter, and should be the same for each user.

String length. If users are left to their own devices to choose passwords, some may choose short strings, while others choose longer strings. If this happens, as it has in experiments where passwords were self-selected, then any distinctiveness detected amongst users cannot be attributed solely to differences among user typing rhythms; the distinctions could have been caused by differences in string lengths that the users typed, or by intrinsic characteristics that cause more variability in one length than in another. So, we don't know if the experimental results are based on user differences or on length differences. To remove this confound, the experimenter should ensure that all participants type same-length strings.

Typing expertise and practice. Everyone has some amount of typing expertise, ranging roughly from low to high. Expertise comes from practice, and the more you practice, the better you get. This pertains to typing just as much as it pertains to piano playing. Two things happen when someone has become practiced at typing a password. First, the total amount of time to type the password decreases; second, the time variation with which particular letter pairs (digrams) are typed diminishes. It takes, on average, about 214 repetitions of a ten-character password to attain a level of expertise such that typing doesn't change by more than 1 millisecond on average (less than 0.1%) over the total time (about 3–5 seconds) taken to type a password. At this level of practice it can be safely assumed that everyone's typing is stable; that is, it's not changing significantly. Due to this stability, it is safe to compare typists using keystroke biometrics. A classifier will be able to distinguish among a group of practiced typists, and will have a particular success rate (often in the region of 95–99%).

But what if, as in some studies, the level of expertise among the subjects ranges from low to high, with some people very practiced and others hardly at all? If practiced typists are consistent, with low variation across repeated typings, but unpracticed typists are inconsistent with high variability, then it would be relatively easy for a classifier to distinguish users in such groups from one another. This could make classification outcomes more optimistic than they really are, making them misleading at best. In one study 25 people were asked to type a password 400 times. Some people in the study did this, but others typed the password only 150 times, putting a potentially large expertise gap between these subjects. No matter what the outcome if everyone had been at the same level of expertise, it's easy to see that the classification results would likely be quite different than if there was a mixture of practice levels among the subjects. This is an example of a lack of internal validity, where the confound of differential expertise or practice is operating. There is no way that the classifier results can be attributed solely to users' typing rhythms alone; they are confounded with level of practice.

Instructions to typists. In any experiment there needs to be a protocol by which the experiment is carried out. This protocol should be followed assiduously, lest errors creep into the experiment whilst the researcher is unaware. Here we give two examples in which instructions to subjects are important.

First, in our own experience, we had told subjects to type the password normally, as if they were logging in to their own computer. This should be straightforward and simple, but it's not. We discovered that some subjects were typing with extraordinary quickness. When we asked those people if that's how they typed every day, they said no—they thought that the purpose of our experiment was to see who could type the fastest or the most accurately, even though we had never said that. This probably happened because we are a university laboratory, and it's not unusual in university experiments (especially in psychology) to have their true intentions disguised from the participant; otherwise the participant may game the experiment, and hence ruin it. People in our experiment assumed that we had a hidden agenda (we didn't), and the people responded to what they thought was the true agenda by typing either very quickly or very carefully or both. When we discovered this, we changed our instructions to tell subjects explicitly that there was no hidden agenda, and that we really meant it when we said that we were seeking their normal, everyday typing behavior. After the instructions were changed to include this, we no longer observed the fast and furious typing behavior that had drawn our attention in the first place. If we had not done this, then we would have left an internal

invalidity in the experiment; our results would have been confounded with normal typing by some and abnormally fast typing by others. Naturally, a classifier would be able to distinguish between fast and slow typists, thereby skewing the outcomes unrealistically.

Second, if there is no written protocol by which to conduct an experiment, and by which to instruct participants as to what they are being asked to do, there is a tendency for the experimenter to ad lib the instructions. While this might be fine, what can happen in practice is that the experimenter will become aware of a slightly better way to word or express the instructions, and will slightly alter the instructions for the next subject. This might slightly improve things for that subject. However, for the subject after that, the instructions might change again, even if ever so slightly. As this process continues, there will come a point at which some of the later subjects are receiving instructions that are quite different from those received by the earlier subjects. This means that two different sets of instructions were issued to subjects, and these subjects may have responded in two different ways, leading to a confound. Whatever the classification outcomes might be, they cannot be attributed solely to differences in user typing rhythms; they might have been due to differences in instructions as well, and we can't tease them apart. Hence it is important not only to have clear instructions, but also to have them in writing so that every subject is exposed to exactly the same set of instructions.

6. What's the solution for all these problems?

All of the problems discussed so far are examples of threats to validity, and internal validity in particular. The confounds we've identified can render an experiment useless, and in those circumstances not only has time and money been wasted, but any published results run a substantial risk of misleading the readership. For example, if a study claims 99.9% correct classification of users typing passwords, that's pretty good; perhaps we can consider the problem solved. But if that 99.9% was achieved because some confound, such as typing expertise, artificially enhanced the results, then we would have reached an erroneous conclusion, perhaps remaining unaware of it. This is a serious research error; in this section we offer some ways to avoid the kinds of problems caused by invalidity.

Control. We use the term "control" to mean that something has been done to mitigate a potential bias or confound in an experiment. For example, if an experimental result could be explained by more than one causal mechanism, then we would need to control that mechanism so that only one cause could be attributed to the experimental outcome. As an example, the length of the password should be controlled so that everyone types a password of the same length; that way, length will not be a factor in classifying typing vectors. A second example would be to control the content of the password, most simply by having every participant type the same password. In doing this, we would be more certain that the outcome of the experiment would be influenced only by differences in people's typing rhythms, and not by password length or content. Of course while effecting control in this way makes the experiment internally valid, it doesn't reflect how users in the real world choose their passwords; certainly they don't all have the same password. But the goal of this experiment is to determine the extent to which individuals have unique typing rhythms, and in that case tight experimental control is needed to isolate all the extraneous factors that might confound the outcome. Once it's determined that people really do have unique typing rhythms that are discriminable, then we can move to the real world with confidence.

Repeatability and reproducibility (again). We earlier mentioned two important concepts: repeatability—the extent to which an experimenter can obtain the same measurements or outcomes when he repeats the experiment in his own laboratory—and reproducibility, which strives for the same thing, but when different experimenters in other laboratories, using similar but physically different apparatus, obtain the same results as the original experimenters did. If we strive to make an experiment repeatable, it means that we try hard to make the same measures each time. To do this successfully requires that all procedures are well defined so that they can be repeated exactly time after time. Such definitions are sometimes called operational definitions, because they specify a measurement in terms of the specific operations used to obtain it. For example, when measuring people's height, it's important that everyone do it the same way. An operational definition for someone's height would specify exactly the procedure and apparatus for taking such

measurements. The procedure should be written so that it can be followed exactly every time. Repeatability can be ensured if the experiment's measurements and procedures are operationally defined and followed assiduously. Reproducibility can be ensured by providing those operational details when reporting the experiment in the literature, thereby enabling others to follow the original procedures.

Discovering confounds. There is no easy way to discover the confounds lurking in an experimental procedure. It requires deep knowledge of the domain and the experiment being conducted, and it requires extensive thought as to how various aspects of the experiment may interact. One approach is to trace the signal of interest (in our case, the keystroke timings and the user behaviors) from their source to the point at which they are measured or manifested. For keystroke timings, the signal begins at the scan matrix in the keyboard, traveling through the keyboard encoder, the keyboard-host interface (e.g., PS2, USB, wireless, etc.), the keyboard controller in the operating system (which is in turn influenced by the scheduler), and finally to the timestamping mechanism, which is influenced by the particular clock being used. At each point along the way, it is important to ask if there are any possible interactions between these waypoints and the integrity of the signal. If there are, then these are candidates for control. For example, keyboard signals travel differently through the PS2 interface than they do through the USB interface. This difference suggests that only one type of keyboard interface be used—either PS2 or USB, but not both. Otherwise, part of the classification accuracy would have to be attributed to the different keyboard interfaces. A similar mapping procedure would ask about aspects of the experiment that would influence user typing behavior. We have already given the example of different types of keyboards causing people to type differently. Countering this would be done simply by using only one type of keyboard.

Method section. A method section in a paper is the section in which the details are provided regarding how the experiment was designed and conducted. Including a method section in an experimental paper has benefits that extend to both reader and researcher. The benefit to the reader is that he can see exactly what was done in the experiment, and not be left to wonder about details that could affect the outcome. For example, saying how a set of experiment participants was recruited can be important; if some were recruited outside the big-and-tall shop, it could constitute a bias in that these people are likely to have large hands, and large-handed people might have typing characteristics that make classification artificially effective or ineffective. If this were revealed in the method section of a paper, then a reader would be aware of the potential confound, and could moderate his expectations on that basis. If the reader were a reviewer, the confound might provoke him to ask the author to make adjustments in the experiment.

For the experimenter the method section has two benefits. First, the mere act of writing the method section can reveal things to the experimenter that were not previously obvious. If, in the course of writing the section, the experimenter discovers an egregious bias or flaw in the experiment, he can choose another approach, he can relax the claims made by the paper, or he can abandon the undertaking to conduct the experiment again under revised and more favorable circumstances. If the method section is written before the experiment is done—as a sort of planning exercise—the flaws will become apparent in time for the experimental design to be modified in a way that eliminates the flaw or confound. This will result in a much better experiment, whose outcome will stand the test of time.

Pilot studies. Perhaps the best way to check your work is to conduct a pilot study—a small-scale preliminary test of procedures and measurement operations—to shake any unanticipated bugs out of an experiment, and to check for methodological problems such as confounded variables. Pilot studies can be very effective in revealing problems that, at scale, would ruin an experiment. It was through a pilot study that we first understood the impact of instructions to subjects, and subsequently adjusted our method to avoid the problems encountered (previously discussed). If there had been no pilot, we would have discovered the problem with instructions anyway, but we could not have changed the instructions in the middle of the experiment, because then we'd have introduced the confound of some subjects having heard one set of instructions, and other subjects having heard a different set; the classification outcome could have been attributed to the differences in instructions as well as to differences amongst typists.

7. Conclusion

We have shown how several very simple oversights in the design and conduct of an experiment can result in confounds and biases that may invalidate experimental outcomes. If the details of an experiment are not fully described in a method section of the paper, there is a risk that the flaws will never be discovered, with the consequence that we come away thinking that we've learned a truth (that isn't true) or we've solved a problem (that isn't really solved). Other researchers may base their studies on flawed results, not knowing about the flaws because there was no information provided that would lead to a deep understanding of how the experiment was designed and carried out. Writing a method section can help experimenters avoid invalidities in experimental design, and can help readers and reviewers determine the quality of the undertaking.

Of course there are still other things that can go wrong. For example, even if you have ensured that your methods and measurements are completely valid, the chosen analysis procedure could be inappropriate for the undertaking. At least, however, you'll have confidence that you won't be starting out with invalid data.

While the confounding issues discussed here apply to an easily-understood domain like keystroke biometrics, they were nevertheless subtle, and have gone virtually unnoticed in the literature for decades. Your own experiments, whether in this domain or another, are likely to be just as susceptible to confounding and methodological errors, and their consequences just as damaging. We hope that this paper has raised the collective consciousness so that other researchers will be vigilant for the presence and effects of methodological flaws, and will do their best to identify and mitigate them.

Richard Feynman, the 1965 Nobel Laureate in physics, said, "The principle of science, the definition almost, is the following: The test of all knowledge is experiment. Experiment is the sole judge of scientific 'truth'" [2]. Truth is separated from fiction by demonstration—by experiment. In doing experiments, we want to make claims about the results. For those claims to be credible, the experiments supporting them need first to be free of the kinds of methodological errors and confounds presented here.

References

[1] Bryan, W.L., Harter, N.: Studies in the physiology and psychology of the telegraphic language. Psychological Review 4(1), 27–53 (1897)

[2] Feynman, R.P., Leighton, R.B., Sands, M.: The Feynman Lectures on Physics, vol. 1, p. 1–1. Addison-Wesley, Reading (1963)

[3] Killourhy, K., Maxion, R.: The effect of clock resolution on keystroke dynamics. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 331–350. Springer, Heidelberg (2008)

[4] Killourhy, K.S., Maxion, R.A.: Comparing anomaly-detection algorithms for keystroke dynamics. In: IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009), pp. 125–134. IEEE Computer Society Press, Los Alamitos (2009)

[5] Maxion, R.A., Killourhy, K.S.: Keystroke biometrics with number-pad input. In: IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010), pp. 201–210. IEEE Computer Society Press, Los Alamitos (2010)

[6] Microsoft Developer Network: EVENTMSG structure (2008), http://msdn2.microsoft.com/en-us/library/ms644966(VS.85).aspx

[7] Peacock, A., Ke, X., Wilkerson, M.: Typing patterns: A key to user identification. IEEE Security and Privacy 2(5), 40–47 (2004)

[8] Shadish, W.R., Cook, T.D., Campbell, D.T.: Experimental and Quasi-Experimental Designs for Generalized Causal Inference. Houghton Mifflin, Boston (2002)

[9] Taylor, B.N., Kuyatt, C.E.: Guidelines for evaluating and expressing the uncertainty of NIST measurement results. NIST Technical Note, 1994 Edition 1297, National Institute of Standards and Technology (NIST), Gaithersburg, Maryland 20899-0001 (September 1994)

About the author

Roy Maxion is a research professor in the Computer Science and Machine Learning Departments at Carnegie Mellon University (CMU). He is also director of the CMU Dependable Systems Laboratory where the range of activities includes computer security, behavioral biometrics, insider detection, usability, and keystroke forensics as well as general issues of hardware/software reliability. In the interest of the integrity of experimental methodologies, Dr. Maxion teaches a course on Research Methods for Experimental Computer Science. He is on the editorial boards of *IEEE Security & Privacy* and the *International Journal of Biometrics,* and is past editor of *IEEE Transactions on Dependable and Secure Computing* and *IEEE Transactions on Information Forensics and Security.* Dr. Maxion is a Fellow of the IEEE.

On bugs and elephants: Mining for science of security

Dusko Pavlovic

1. On security engineering

> A number of blind men came to an elephant. Somebody told them that it was an elephant. The blind men asked, "What is the elephant like?" and they began to touch its body. One of them said: "It is like a pillar." This blind man had only touched its leg. Another man said, "The elephant is like a husking basket." This person had only touched its ears. Similarly, he who touched its trunk or its belly talked of it differently.
>
> ~Ramakrishna Paramahamsa~

Security means many things to many people. For a software engineer, it often means that there are no buffer overflows or dangling pointers in the code. For a cryptographer, it means that any successful attack on the cypher can be reduced to an algorithm for computing discrete logarithms or to integer factorization. For a diplomat, security means that the enemy cannot read the confidential messages. For a credit card operator, it means that the total costs of the fraudulent transactions and of the measures to prevent them are low, relative to the revenue. For a bee, security means that no intruder into the beehive will escape her sting . . .

Is it an accident that all these different ideas go under the same name? What do they really have in common? They are studied in different sciences, ranging from computer science to biology, by a wide variety of different methods. Would it be useful to study them together?

1.1. What is security engineering?

If all avatars of security have one thing in common, it is surely the idea that *there are enemies and potential attackers out there.* All security concerns, from computation to politics and biology, come down to averting the adversarial processes in the *environment* that are poised to subvert the goals of the *system.* There are, for instance, many kinds of bugs in software, but only those that the hackers use are a security concern.

In all engineering disciplines, the system guarantees a functionality, provided that the environment satisfies some assumptions. This is the standard assume-guarantee format of the engineering correctness statements. Such statements are useful when the environment is passive so that the assumptions about it remain valid for a while. The essence of security engineering is that System and Environment face off as opponents, and Environment actively seeks to invalidate System's assumptions.

Security is thus an adversarial process. In all engineering disciplines, failures usually arise from some engineering errors. In security, failures arise in spite of compliance with the best engineering practices of the moment. Failures are the first-class citizens of security. For all major software systems, we normally expect security updates, which usually arise from attacks and often inspire them.

1.2. Where did security engineering come from?

The earliest examples of security technologies are found among the earliest documents of civilization. Figure 1, on the following page, shows security tokens with a tamper protection technology from almost 6,000 years ago. Figure 2 depicts the situation where this technology was probably used. Alice has a lamb and Bob has built a secure vault, perhaps with multiple security levels, spacious enough to store both Bob's and Alice's assets. For each of Alice's assets deposited

FIGURE 1. Tamper protection (bulla envelope with 11 plain and complex tokens inside) from the Near East, circa 3700–3200 BC. (The Schøyen Collection MS 4631. ©The Schøyen Collection, Oslo and London. Available at: www.schoyencollection.com.)

in the vault, Bob issues a clay token with an inscription identifying the asset. Alice's tokens are then encased into a bulla—a round, hollow envelope of clay—that is then baked to prevent tampering. When she wants to withdraw her deposits, Alice submits her bulla to Bob; he breaks it, extracts the tokens, and returns the goods. Alice can also give her bulla to Carol, who can also submit it to Bob to withdraw the goods, or pass it on to Dave. Bullae can thus be traded and facilitate an exchange economy. The tokens used in the bullae evolved into the earliest forms of money; and the inscriptions on them led to the earliest numeral systems, as well as to Sumerian cuneiform script, which was one of the earliest alphabets. Security thus predates literature, science, mathematics, and even money.

1.3. Where is security engineering going?

Through history, security technologies evolved gradually, serving the purposes of war and peace, protecting public resources and private property. As computers pervaded all aspects of social life, security became interlaced with computation, and security engineering came to be closely related with computer science. The developments in the realm of security are nowadays inseparable from the developments in the realm of computation. The most notable such development is, of course, cyberspace.

A brief history of cyberspace. In the beginning, engineers built computers and wrote programs to control computations. The platform of computation was the computer, and it was used to execute algorithms and calculations, allowing people to discover, for example, fractals, and to invent compilers that allowed them to write and execute more algorithms and more calculations more efficiently. Then the operating system became the platform of computation, and software was developed on top of it. The era of personal computing and enterprise software broke out. And then the Internet happened, followed by cellular networks, and wireless networks, and ad hoc networks, and mixed networks. Cyberspace emerged as the distance-free

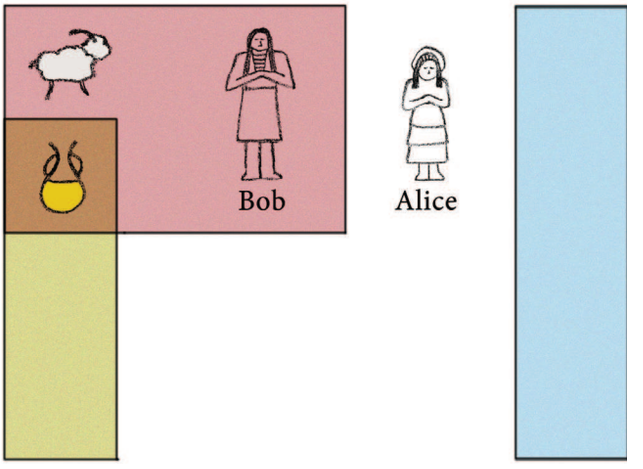


FIGURE 2. To withdraw her sheep from Bob's secure vault, Alice submits a tamper-proof token, like those shown in figure 1.

space of instant, costless communication. Nowadays, software is developed to run in cyberspace.

The Web is, strictly speaking, just a software system, albeit a formidable one. A botnet is also a software system. As social space blends with cyberspace, many social (business, collaborative) processes can be usefully construed as software systems that run on social networks as hardware. Many social and computational processes become inextricable. Table 1 summarizes the crude picture of the paradigm shifts that led to this remarkable situation.

TABLE 1. Paradigms of computation

	Ancient Times	Middle Ages	Modern Times
Platform	computer	operating system	network
Applications	Quicksort, compiler	MS Word, Oracle	WWW, botnets
Requirements	correctness, termination	liveness, safety	trust, privacy
Tools	programming languages	specification languages	scripting languages

But as every person got connected to a computer, and every computer to a network, and every network to a network of networks, computation became interlaced with communication and ceased to be programmable. The functioning of the web and of web applications is not determined by the code in the same sense as in a traditional software system; after all, web applications do include the human users as a part of their runtime. The fusion of social and computational processes in cybersocial space leads to a new type of information processing, where the purposeful program executions at the network nodes are supplemented by spontaneous data-driven evolution of network links. While the network emerges as the new computer, data and metadata become inseparable, and a new type of security problems arises.

A brief history of cybersecurity. In early computer systems, security tasks mainly concerned sharing of the computing resources. In computer networks, security goals expanded to include information protection. Both computer security and information security essentially depend on a clear distinction between the secure areas and the insecure areas, separated by a security perimeter. Security engineering caters for computer security and for information security by providing the tools to build the security perimeter. In cyberspace, the secure areas are separated from the insecure areas by the "walls" of cryptography, and they are connected through the "gates" of cryptographic protocols.

But as networks of computers and devices spread through physical and social spaces, the distinctions between the secure and the insecure areas become blurred. And in such areas of cybersocial space, where information processing does not yield to programming and cannot be secured by cryptography and protocols, security cannot be assured by engineering methodologies alone. The methodologies of data mining and classification, needed to secure such areas, form a bridge from information science to a putative security science.

2. On security science

> It is the aim of the natural scientist to discover mathematical theories, formally expressed as predicates describing the relevant observations that can be made of some [natural] system. . . . The aim of an engineer is complementary to that of the scientist. He starts with a specification, formally expressible as a predicate describing the desired observable behaviour. Then . . . he must design and construct a product that meets that specification.
>
> ~Tony Hoare~

The preceding quote was the first paragraph in one of the first papers on formal methods for software engineering, published under the title "Programs are predicates." Following this slogan, software has been formalized by logical methods and viewed as an engineering task ever since. But computation evolved, permeated all aspects of social life, and came to include not just the purposeful program executions, but also spontaneously evolving network processes. Data and metadata processing became inseparable. In cyberspace, computations are not localized at network nodes, but also propagate with nonlocal data flows and with the evolution of network links. While the local computations remain the subject of software engineering, network processes are also studied in the emerging software and information sciences, where the experimental validation of mathematical models

has become the order of the day. Modern software engineering is therefore coupled with an empiric software science, as depicted in figure 3. In a similar way, modern security engineering needs to be coupled with an empiric security science.

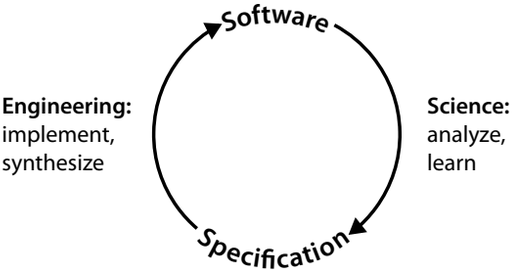


FIGURE 3. Conceptualization loop: The life cycle of computation.

2.1. Why security science?

Conjoining cyber, physical, and social spaces by networks gives rise to new security problems that combine computational, physical, and social aspects. They cross the boundaries of the disciplines where security was studied before, and require new modeling tools, and a new, unified framework, with a solid scientific foundation, and empiric methods to deal with the natural and social processes on which security now depends. In many respects, a scientific foundation for the various approaches to security would have been beneficial even before; but now it became necessary.

Let us have a closer look at the paradigm shift to postmodern cybersecurity in table 2. It can be illustrated as the shift from figure 4 to figure 5. The fortress in figure 4 represents the static, architectural view of security. A fortress consists of walls and gates separating the secure area within from the insecure area outside. The boundary between these two areas is the security perimeter. The secure area may be further subdivided into areas of higher security and areas of lower security. These intuitions extend into cyberspace, where crypto systems and access controls can be viewed as the walls, preventing the undesired traffic; whereas, authentication protocols and authorization mechanisms can be construed as the gates, allowing the desired traffic. But as every fortress owner knows, the walls and the gates are not enough for security; you also need weapons, soldiers, and maybe even some detectives and judges. They take care of the dynamic aspects of security. Dynamic security evolves through social processes, such as trust, privacy, reputation, or influence. The static and dynamic aspects depend on each other. For example, the authentication on the gates is based on some credentials intended to prove that the owner is honest. These credentials may be based on some older credentials, but down the line a first credential must have resulted from a process of trust building or from a trust decision, whereby the principal's honesty was accepted with no credentials. The word *credential* has its root in Latin *credo,* which means "I believe."

The attacks mostly studied in security research can be roughly divided into cryptanalytic attacks and protocol attacks. They are the cyber versions of the simple frontal attacks on the walls and the gates of a fortress. Such attacks are static in the sense that the attackers are outside, the defenders inside, and the two are easily distinguished. The dynamic attacks come about when some attackers penetrate the security perimeter and attack from within, as in figure 5. They may even blend with the defenders and become spies. Some of them may build up trust and infiltrate the fortress earlier, where they wait as moles. Some of the insiders may defect and become attackers. The traitors and the spies are the dynamic attackers; they use the vulnerabilities in the process of trust. To deter them, all cultures reserve for the breaches of trust the harshest punishments imaginable; Dante, in his description of Hell, places the traitors into the deepest, Ninth Circle. As a dynamic attack, treason was always much easier to punish than to prevent.

In cybersecurity, a brand new line of defense against dynamic attacks relies on predictive analytics, based on mining the data gathered by active or passive

TABLE 2. Paradigms of security

	Middle Ages	Modern Times	Postmodern Times
Space	computer center	cyberspace	cybersocial space
Assets	computing resources	information	public and private resources
Requirements	availability, authorization	integrity, confidentiality	trust, privacy
Tools	locks, tokens, passwords	cryptography, protocols	mining and classification

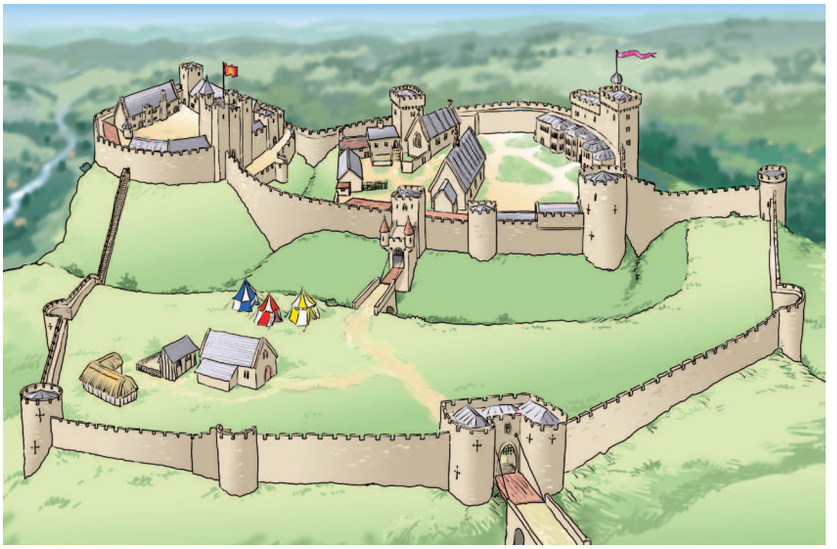


FIGURE 4. Static security: Multilevel architecture. (Illustration by Mark Burgess at www.markburgess.co.uk.)

observations, network probes, honeypots, or direct interactions. It should be noted that the expanding practices of predictive modeling are not engineering methodologies, geared toward building some specified systems, but the first simple tools of a security science, recognizing security as a process.

2.2. What is security science?

Although the security environment maliciously defies any system's assumptions that it can, security engineering still pursues its tasks strictly within the framework of the assume-guarantee methods. Indeed, to engineer a system, we must frame an environment for it; to guarantee system behavior, we must assume the environment behavior; to guarantee system security, we must specify an attacker model. That is the essence of the engineering approach. Following that approach, the cryptographic techniques of security engineering are based on the fixed assumption that the environment is computationally limited and cannot solve certain hard problems. (Defy that, Environment!)

But sometimes, as we have seen, it is not realistic to assume even that there is a clear boundary between the system and the environment. Such situations have become pervasive with the spread of networks supporting not only social, commercial, and collaborative applications, but also criminal and terrorist organizations. When there is a lot going on, you cannot be sure who is who. In large networks, with immense numbers of processes, the distinction between the system and the environment becomes meaningless, and the engineering *assume-guarantee approach* must be supplemented by the *analyze-adapt approach* of science. The task of the analyze-adapt approach of science is to recover the distinction between system and environment—whenever possible, albeit as a dynamic variable—and to adaptively follow its evolution. Similar situations, where engineering interventions are interleaved with scientific analyses, arise not only in security—where they elicit security science to support security engineering—but also, for example, in the context of health—where they elicit medical science to support health care. And just as health is not achieved by isolating the body from the external world, but by supporting its internal defense mechanisms, security is not achieved by erecting fortresses, but by supporting

FIGURE 5. Security dynamics: Threats within.

dynamic defenses, akin to the immune response. While security engineering provides blueprints and materials for static defenses, it is the task of security science to provide guidance and adaptation methods for dynamic defenses.

In general, science is the process of understanding the environment, adapting the system to it, changing the environment by the system, adapting to these changes, and so on. Science is thus an ongoing dialog of the system and the environment, separated and conjoined along the ever-changing boundaries. Dynamic security, on the other hand, is an ongoing battle between the ever-changing teams of attackers and defenders. Only scientific probing and analyses of this battle can tell who is who at any particular moment.

In summary, if security engineering is a family of methods to keep the attackers out, security science is a family of methods to catch the attackers once they get in.

It may be interesting to note that these two families of methods, viewed as strategies in an abstract security game, turn out to have opposite winning odds. It is often observed that the attackers only need to find one attack vector to enter the fortress, whereas the defenders must defend all attack vectors to prevent them. But when the battle switches to the dynamic mode and the defense moves inside, then the defenders only need to find one marker to recognize and catch the attackers; whereas, the attackers must cover all their markers. This strategic advantage is also the critical aspect of the immune response, where the invading organisms are purposely sampled and analyzed for chemical markers. In security science, this sampling and analyses take the form of data mining.

2.3. Where to look for security science?

The germs of a scientific approach to security, with data gathering, statistical analyses, and experimental validation, are already present in many intrusion detection and antivirus systems, as well as in spam filters and some firewalls. Such systems use measurable inputs and have quantifiable performance and model accuracy and thus conform to the basic requirements of the scientific method. The collaborative processes for sharing data, comparing models, and retesting and unifying results complete the social process of scientific research.

However, a broader range of deep security problems is still awaiting applications of a broader range of powerful scientific methods that are available in this realm. At least initially, the statistical methods of security science will need to be borrowed from information science. Security, however, imposes special data analysis requirements, some of which have been investigated in the existing work and led to novel approaches. In the long run, security science will undoubtedly engender its own domain-specific data analysis methods.

In general, security engineering solutions are based on security infrastructure: Internet protocol security (IPSec) suites, Rivest-Shamir-Adleman (RSA) systems, and elliptic curve cryptography (ECC) provide typical examples. In contrast, security science solutions emerge where the available infrastructure does not suffice for security. The examples abound—a mobile ad hoc network (MANET), for example, is a network of nodes with no previous contacts, direct or indirect, and thus no previous infrastructure. Although advanced MANET technologies have been available for more than 15 years, secure MANETs are still a bit of a holy grail. Device pairing, social network security, and web commerce security also require secure ad hoc interactions akin to the social protocols that regulate new encounters in social space. Such protocols are invariably incremental and accumulating, analyzing and classifying the data from multiple channels until a new link is established or aborted. Powerful data-mining methods have been developed and deployed in web commerce and financial security, but they are still awaiting systematic studies in noncommercial security research and systematic applications in noncommercial security domains.

3. Summary

Security processes are distributed, subtle, and complex, and there are no global observers. Security is like an elephant, and we are like the blind men touching its body. For the cryptographers among us, the security elephant consists of elliptic curves and of integers with large factors. Many software engineers among us derive their view of the security elephant entirely from their view of the software bugs flying around it.

Beyond and above all of our partial views is the actual elephant—people cheating each other, stealing secrets and money, forming online gangs and terrorist networks. There is a whole wide world of social

processes of attacking and defending the assets by methods beyond the reach of security engineering. Such attacks and fraud cannot be debugged or programmed away; they cannot be eliminated by cryptography, protocols, or policies. Security engineering defers such attacks to the marginal notes about "social engineering."

However, since these attacks nowadays evolve in networks, the underlying social processes can be observed, measured, analyzed, understood, validated, and even experimented with. Security can be improved by security science, combining and refining the methods of information sciences, social sciences, and computational sciences. 

Acknowledgements

Just like security, science of security also means many things to many people. I have presented one view of it, not because it is the only one I know, but mainly because it is the simplest one that I could think of, and maybe the most useful one. But some of my good friends and collaborators see it differently, and I am keeping an open mind. I am grateful to Brad Martin and Robert Meushaw for interesting conversations and, above all, for their initiative in this area.

About the author

Dusko Pavlovic is a professor of information security at Royal Holloway, University of London. He received his PhD in mathematics at the Utrecht University in 1990. His interests evolved from research in pure mathematics and theoretical computer science, through software design and engineering, to problems of security and network computation. He worked in academia in Canada, the United Kingdom, and the Netherlands, and in software research and development in the United States. Besides the chair in information security at Royal Holloway, he currently holds a chair in security protocols at University of Twente, and a visiting professorship at University of Oxford. His research projects are concerned with extending the mathematical methods of security beyond the standard cryptographic models toward capturing the complex phenomena that arise from physical, economic, and social aspects of security processes.

Programming language methods for compositional security |

Anupam Datta and
John C. Mitchell

Divide-and-conquer is an important paradigm in computer science that allows complex software systems to be built from interdependent components. However, there are widely recognized difficulties associated with developing divide-and-conquer paradigms for computer security; we do not have principles of compositional security that allow us to put secure components together to produce secure systems. The following article illustrates some of the problems and solutions we have explored in recent research on compositional security, compares them to other approaches explored in the research community, and describes important remaining challenges.

1. Introduction

Compositional security is a well-recognized scientific challenge [1]. Contemporary systems are built up from smaller components, but even if each component is secure in isolation, a system composed of secure components may not meet its security requirements—an adversary may exploit complex interactions between components to compromise security. Attacks using properties of one component to subvert another have shown up in practice in many different settings, including network protocols and infrastructure [2, 3, 4, 5, 1], web browsers and infrastructure [6, 7, 8, 9, 10], and application and systems software and hardware [11, 12, 13].

A theory of compositional security should identify *relationships* among systems, adversaries, and properties, such that precisely defined operations over systems and adversaries preserve security properties. It should *explain* known attacks, *predict* previously unknown attacks, and *inform* design of new systems. The theory should be *general*—it should apply to a wide range of systems, adversaries, and properties. Guided by these desiderata, we initiated an investigation of compositional security in the domain of security protocols with the Protocol Composition Logic (PCL) project [14, 15, 16]. Building on these results, we then developed general secure composition principles that *transcend specific application domains* (for example, security protocols, access control systems, web

platform) in the Logic of Secure Systems (LS^2) project [17]. These theories have been applied to explain known attacks, predict previously unknown attacks, and inform the design of practical protocols and software systems [12, 4, 18, 3, 19, 20, 21].

In both projects, we addressed two basic problems in compositional security: nondestructive and additive composition.

Nondestructive composition ensures that if two system components are combined, then neither degrades the security properties of the other. This is particularly complicated when system components share state. For example, if an alternative mode of operation is added to a protocol, then some party may initiate a session in one mode and simultaneously respond to another session in another mode, using the same public key (an example of shared state) in both. Unless the modes are designed not to interfere, there may be an attack on the multimode protocol that would not arise if only one mode were possible. In a similar example, new attacks became possible when trusted computing systems were augmented with a new hardware instruction that could operate on protected registers (an example of shared state) previously accessible only through a prescribed protocol [12].

Additive composition supports a combination of system components in a way that accumulates security properties. Combining a basic key exchange protocol with an authentication mechanism to produce a protocol for authenticated key exchange provides one example of additive composition [15]. Systematically adding cryptographic operations to basic authentication protocols to provide additional properties such as identity protection provides another example of additive composition [22].

Both additive and nondestructive compositions are important in practice. If we want a system with the positive security features of two components, *A* and *B*, we need nondestructive composition conditions to be sure that we do not lose security features we want, and we need additive composition conditions to make sure we get the advantages of *A* and *B* combined.

Before turning to a high-level presentation of technical aspects of nondestructive and additive composition in PCL and LS^2, we present two concrete examples that illustrate how security properties fail to be preserved under composition (that is, both examples are about the failure of nondestructive composition). We also compare our composition methods to three related approaches—compositional reasoning for correctness properties of systems [23, 24], the universal composability framework [25, 26], and a refinement type system for compositional type-checking of security protocols [27]. Finally, we describe directions for future work.

2. Two examples

While these protocol examples are contrived, the phenomena they illustrate are not: It is possible for one component of a system to expose an interface to the adversary that does not affect its own security but compromises the security of other components. Later, we will describe two general principles of compositional security that could be used to design security protocols and other kinds of secure software systems while avoiding the kind of insecure interaction illustrated by these examples.

Example 1: Authentication failure. The following two protocols use digital signatures. The first protocol provides one-way authentication when used in isolation; however, this property is not preserved when the second protocol is run concurrently.

- *Protocol 1.1.* Alice generates a fresh random number *r* and sends it to Bob. Upon receiving such a message, Bob replies to the sender of the message (as recorded in the message) with his signature over the fresh random number and

the sender's name—that is, if Bob receives the message with the random number *r* from sender *A*, then Bob replies with his signature over *r* and *A*. This protocol guarantees a form of one-way authentication: After sending the first message to Bob and then receiving Bob's second message, Alice is guaranteed that Bob received the first message that she sent to him and then sent the second message and intended it for her.

- *Protocol 1.2.* Upon receiving any message *m*, Bob signs it with his private signing key and sends it out on the network.

When the two protocols are run concurrently, protocol 1.1 no longer provides one-way authentication: Alice cannot be certain that Bob received her first message and intended the signed message for her as part of the execution of this protocol; it could very well be that Bob produced the signature as part of protocol 1.2 in response to an adversary *M* who intercepted Alice's message and used it to start a session of protocol 1.2 with Bob.

Example 2: Secrecy failure. Using network protocols as an illustration, here are two secure, unidirectional protocols for communication between Alice and Bob. Both involve public key cryptography, in which two different keys are used for encryption and decryption, and the encryption key may be distributed publicly.

- *Protocol 2.1.* In this protocol, for communication from Alice to Bob, Alice sends a message to Bob by encrypting it with Bob's public encryption key. As part of each message, in order to make our example illustrate the general point, Alice also reveals her secret decryption key, making public-key encryption to Alice insecure.
- *Protocol 2.2.* This protocol is the same as the previous one (that is, protocol 2.1), but in reverse: Bob communicates to Alice by encrypting messages using Alice's public key and revealing his own private decryption key.

Both protocol 2.1 and 2.2 are secure when used by themselves: If Bob sends Alice a message encrypted with Alice's public key, then only Alice can decrypt and read the message. However, it should be clear that composing these two protocols to communicate between Alice and Bob in both directions is completely insecure because when Alice sends Bob a message, she leaks her private key, and when Bob communicates to Alice, he leaks his private key. After at least one message in each direction, both public keys have been leaked and any eavesdropper on the network can decrypt and read all the messages.

3. Two principles of secure composition

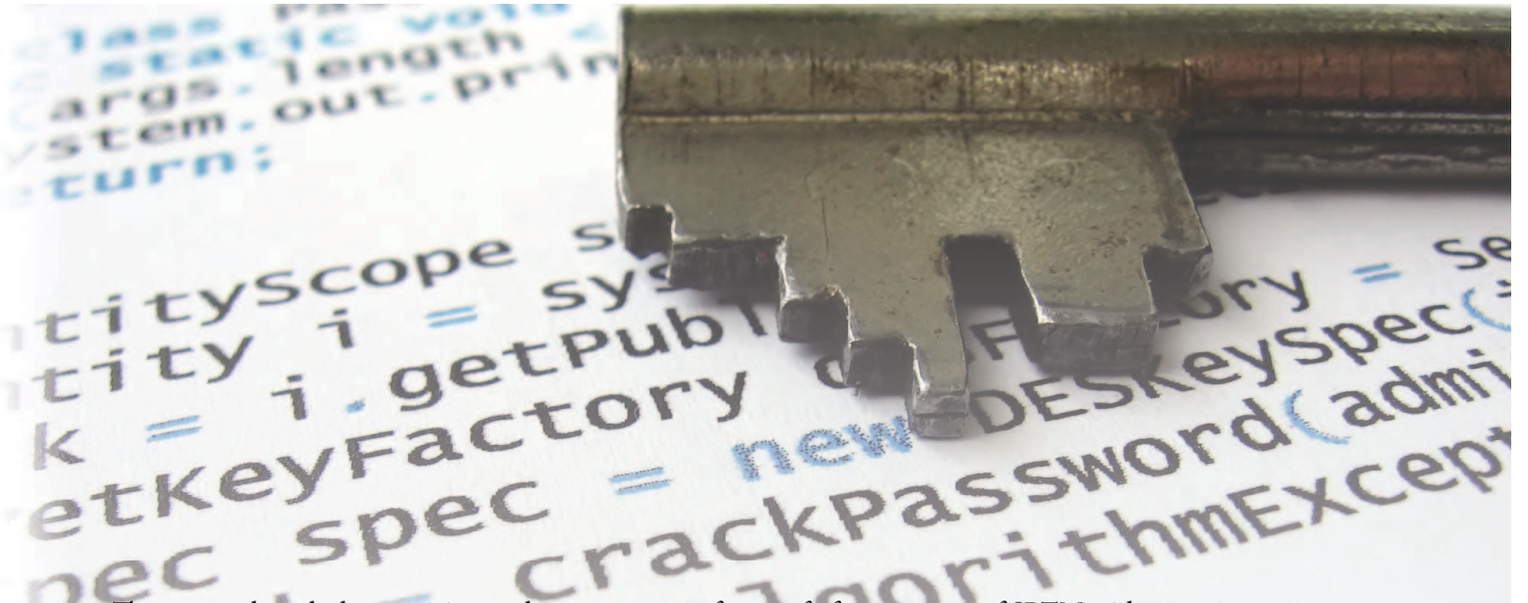
In the following, we describe two principles of secure composition, and we use these principles to explain the examples of insecure composition in the previous section.

3. 1. Principle 1: Preserving invariants of system components

The central idea behind this principle is that the security property of a system component is preserved under composition if every other component respects invariants used in the proof of security of the component in the face of attack. In example 1, the only relevant invariant for the authentication property of protocol 1.1 is of the following form: "If an honest[a] principal signs a message of the form $< r, A >$, then he must have previously received *r* in a message with *A* as the identifier for the sender." This invariant is not preserved by protocol 1.2, as demonstrated by the attack described in the previous section, leading to a failure of nondestructive composition.

To illustrate the generality of this principle, we briefly discuss a published analysis of the widely deployed Trusted Computing Group (TCG) technology using this principle [12], and we discuss the consequent discovery of a real incompatibility between an existing standard protocol for attesting the integrity of the software stack to a remote party and a newly added hardware instruction. Machines with trusted computing abilities include a special, tamper-proof hardware called the Trusted Platform Module or TPM, which contains protected append-only registers to store measurements (that is, hashes) of programs loaded into memory and a dedicated coprocessor to sign the contents of the registers with a unique hardware-protected key. The protocol in question, called Static Root of Trust Measurement (SRTM), uses this hardware to establish the integrity of the software stack on a machine to a trusted remote third

a. A principal is honest if he does not deviate from the steps of the protocol.



party. The protocol works by requiring each program to store, in the protected registers, the hash of any program it loads. The hash of the first program loaded into memory, usually the boot loader, is stored in the protected registers by the booting firmware, usually the basic input/output system (BIOS). The integrity of the software stack of a machine following this protocol can be proved to a third party by asking the coprocessor to sign the contents of the protected registers with the hardware-protected key, and sending the signed hashes of loaded programs to the third party. The third party can compare the hashes to known ones, thus validating the integrity of the software stack.

Note that the SRTM protocol is correct only if software that has not already been measured cannot append to the protected registers. Indeed, this invariant was true in the hardware prescribed by the initial TCG standard and, hence, this protocol was secure then. However, a new instruction, called `latelaunch`, added to the standard in a later extension allows an unmeasured program to be started with full access to the TPM. This violates the necessary invariant- and results in an actual attack on the SRTM protocol: A program invoked with `latelaunch` may add hashes of arbitrary programs to the protected registers without actually loading them. Since the program is not measured, the remote third party obtaining the signed measurements will never detect its presence. An analysis of the protocol using the method outlined here discovered this incompatibility between the SRTM protocol and the `latelaunch` instruction. In the analysis, the TPM instruction set, including `latelaunch`, were modeled as interfaces available to programs. The invariant can be established for all interfaces except `latelaunch`, thus leading to failure of a proof of correctness of SRTM with `latelaunch` and leading to discovery of the actual attack.

This composition principle is related to the form of assume-guarantee reasoning initially proposed by Jones for reasoning about correctness properties of concurrent programs [23]. However, one difference is that, in contrast to Jones' work, we consider preservation of properties of system components under composition in the presence of an active adversary whose exact program (or invariants) is not known. After sketching the technical approach in the next sections, we will explain how we address this additional complexity.

3.2. Principle 2: Secure rely-guarantee reasoning

Inductive security properties (that is, properties which hold at a point of time if and only if they have held at all prior points of time) require a different form of compositional reasoning that builds on prior work on rely-guarantee reasoning for correctness properties of programs [23, 24].

Suppose we wish to prove that property φ holds at all times. First, we identify a set $S = \{T_1, \ldots, T_n\}$ of trusted components relevant to the property and local properties $\Psi_{T1}, \ldots, \Psi_{Tn}$ of these components, satisfying the following conditions:

(1) If φ holds at all time points strictly before any given time point, then each of $\Psi_{T1}, \ldots, \Psi_{Tn}$ holds at the given time point.

(2) If φ does not hold at any time, then at least one of $\Psi_{T1}, \ldots, \Psi_{Tn}$ must have been violated strictly before that time.

The rely-guarantee principle states that under these conditions, if φ holds initially, then φ holds forever.

We return to example 2 to illustrate the application of this principle. In order to prove the secrecy of the encrypted message, it is necessary to prove that the private decryption key is known only to the associated party. If protocol 2.1 (or protocol 2.2) were to run in isolation, the relevant decryption key would indeed be known only to the associated party (Alice or Bob). This can be proved using the rely-guarantee reasoning technique described above and noting that the recipient of the encrypted message never sends out his or her private decryption key and that the other party cannot send it out (assuming that it has not already been sent out). However, when the two protocols are composed in parallel, the proof no longer works because the sender in one protocol is the recipient in the other; thus, we can no longer prove that the recipient's private decryption key is not sent out on the network. Indeed, the composition attack arises precisely because the recipient's private decryption key is sent out on the network.

Another application of the rely-guarantee technique is in proofs of secrecy of symmetric keys generated in network protocols. We explain one instance here—proving that the so called authentication key (AKey) generated during the Kerberos V protocol (a widely used industry standard) becomes known only to three protocol participants [17, 18]: the client authenticated by the key, the Kerberos authentication server (KAS) that generates the key, and the ticket granting server (TGS) to whom the key authenticates the client. At the center of this proof is the property that whenever any of these three participants send out the AKey onto the (unprotected) network, it is encrypted with other secure keys. Proving this property requires induction because, as part of the protocol, the client blindly forwards an incoming message to the TGS. Consequently, the client's outgoing message does not contain the unencrypted AKey because the incoming message does not contain the unencrypted AKey in it. The latter follows from the inductive hypothesis that any network adversary could not have had the unencrypted AKey to send to the client.

Formally, the rely-guarantee framework is instantiated by choosing φ to be the property that any message sent out on the network does not contain the unencrypted AKey. The properties Ψ_T, for components T of the client, KAS, and the TGS model the requirement that the respective components do not send out the AKey unencrypted. Then, the proof of condition (2) of the rely-guarantee framework is trivial, and condition (1) follows from an analysis of the programs of the client, the KAS, and the TGS. The first of these, as mentioned earlier, uses the assumption that φ holds at all points in the past. Note that the three programs are analyzed individually, even though the secrecy property relies on the interactions between them, that is, the proof is compositional.

4. Protocol Composition Logic

Protocol Composition Logic (PCL) [14, 15, 16] is a formal logic for proving security properties of network protocols that use public and symmetric key cryptography. The system has several parts:

- **A simple programming language for defining protocols by writing programs for each role of the protocol.** For example, the secure sockets layer (SSL) protocol can be modeled in this language by writing two programs—one for the client role and one for the server role of SSL. Each program is a sequence of actions, such as sending and receiving messages, decryption, and digital signature verification. The operational semantics of the programming language define how protocols execute concurrently with a symbolic adversary (sometimes referred to as the Dolev-Yao adversary) that controls the network but cannot break the cryptographic primitives.
- **A pre/postcondition logic for describing the starting and ending security conditions for protocol.** For example, a precondition might state that a symmetric key is shared by two agents, and a postcondition might state that a new key exchanged using the symmetric key for encryption is only known to the same two agents.
- **Modal formulas, denoted $\theta[P]_X\phi$, for stating that if a precondition θ holds initially, and a protocol thread X completes the steps P, then the postcondition ϕ will be true afterwards irrespective of concurrent actions by other agents and the adversary.** Typically, security properties of protocols are specified in PCL using such modal formulas.

- **A formal proof system for deriving true modal formulas about protocols.** The proof system consists of axioms about individual protocol actions and inference rules that yield assertions about protocols composed of multiple steps.

One of the important ideas in PCL is that although assertions are written only using the steps of the protocol, the logic is sound in a strong sense: Each provable assertion involving a sequence of actions holds in any protocol run containing the given actions and arbitrary additional actions by a malicious adversary. This approach lets us prove security properties of protocols under attack while reasoning only about the actions of honest parties in the protocol, thus significantly reducing the size of protocol proofs in comparison to other proof methods, such as Paulson's Inductive Method [28].

Intuitively, additive combination is achieved using modal formulas of the form $\theta[P]_A\phi$. For example, the precondition θ might assert that A knows B's public key, the actions P allow A to receive a signed message and verify B's signature, and the postcondition ϕ may say that B sent the signed message that A received. The importance of modal formulas with before-after assertions is that we can combine assertions about individual protocol steps to derive properties of a sequence of steps: If $\phi[P]_A\Psi$ and $\Psi[P']_A\theta$, then $\phi[PP']_A\theta$. For example, an assertion assuming that keys have been successfully distributed can be combined with steps that do key distribution to prove properties of a protocol that distributes keys and uses them.

We ensure one form of nondestructive combination using invariance assertions, capturing the first composition principle described in Section 3. The central assertion in our reasoning system, $\Gamma \vdash \phi[P]_A\Psi$, says that in any protocol satisfying the invariant Γ, the before-after assertion $\phi[P]_A\Psi$ holds in any run (regardless of any actions by any dishonest attacker). Typically, our invariants are statements about principals that follow the rules of a protocol, as are the final conclusions. For example, an invariant may state that every honest principal maintains secrecy of its keys, where honest means simply that the principal only performs actions that are given by the protocol. A conclusion in such a protocol may be that if Bob is *honest* (so no one else knows his key), then after Alice sends and receives certain messages, Alice knows that she has communicated with Bob. Nondestructive combination occurs when two protocols are combined and neither violates the invariants of the other.

PCL also supports a specialized form of secure rely-guarantee reasoning about secrecy properties, capturing the second composition principle in Section 3. In order to prove that the network is safe (that is, all occurrences of the secret on the network appear under encryption with a set of keys $\mathbf{K}$ not known to the adversary), the proof system requires us to prove that assuming that the network is safe, all honest agents only send out "safe" messages, that is, messages from which the secret cannot be extracted without knowing the keys in the set $\mathbf{K}$ [18].

These composition principles have been applied to prove properties of a number of industry standards including SSL/TLS, IEEE 802.11i, and Kerberos V5.

5. Logic of Secure Systems

The Logic of Secure Systems (LS^2) (initially presented in [12]) builds on PCL to develop related composition principles for secure systems that perform network communication and operations on local shared memory as well as on associated adversary models. These principles have been applied to study industrial trusted computing system designs. The study uncovered an attack that arises from insecure composition between two remote attestation protocols (see [12] for details). A natural scientific question to ask is whether one could build on these results to develop general secure composition principles that transcend specific application domains, such as network protocols and trusted computing systems. Subsequent work on LS^2 [17], which we turn to next, answers exactly this question.

Two goals drove the development of LS^2. First, we posit that a general theory of secure composition must enable one to flexibly model and parametrically reason about different classes of adversaries. To develop such a theory, we view a trusted system in terms of the interfaces its various components expose: Larger trusted components are built by connecting interfaces in the usual ways (client-server, call-return, message-passing, etc.). The adversary is confined to some subset of the interfaces, but its program is unspecified and can call those interfaces in ways that are not known a priori. Our focus on interface-confined adversaries thus provides a generic way to model different classes of

adversaries in a compositional setting. For example, in virtual machine monitor-based secure systems, we model an adversarial guest operating system by confining it to the interface exposed by the virtual machine monitor. Similarly, adversary models for web browsers, such as the gadget adversary (an attractive vector for malware today that leverages properties of Web 2.0 sites), can be modeled by confining the adversary to the read and write interfaces for frames guarded by the same-origin policy as well as by frame navigation policies [7]. The network adversary model considered in prior work on PCL and the adversary against trusted computing systems considered in the initial development of LS^2 are also special cases of this interface-confined adversary model. At a technical level, interfaces are modeled as recursive functions in an expressive programming language. Trusted components and adversaries are also represented using programs in the same programming language. Typically, we assume that the programs for the trusted components (or their properties) are known. However, an adversary is modeled by considering all possible programs that can be constructed by combining calls to the interfaces to which the adversary is confined.

Our second goal was to develop compositional reasoning principles for a wide range of classes of interconnected systems and associated interface-confined adversaries that are described using a rich logic. The approach taken by LS^2 uses a logic of program specifications, employing temporal operators to express not only the states and actions at the beginning and end of a program, but also at points in between. This expressiveness is crucial because many security properties of interest, such as integrity properties, are safety properties [29]. LS^2 supports the two principles of secure composition discussed in the previous section in the presence of such interface-confined adversaries. The first principle follows from a proof rule in the logic, and the second principle follows from first-order reasoning in the logic. We refer the interested reader to our technical paper for details [17].

6. Related work

We compare our approach to three related approaches—compositional reasoning for correctness properties of systems [23, 24], the Universal Composability (UC) framework [25, 26], and a refinement type system for compositional type-checking of security protocols [27].

The secure composition principles we developed are related to prior work on rely-guarantee reasoning for correctness properties of programs [23, 24]. However, the prior work was developed for a setting in which all programs are known. In computer security, however, it is unreasonable to assume that the adversary's program is known a priori; rather, we model adversaries as arbitrary programs that are confined to certain system interfaces as explained earlier. We prove invariants about trusted programs and system interfaces that hold irrespective of concurrent actions by other trusted programs and the adversary. This additional generality, which is crucial for the secure composition principles, is achieved at a technical level using novel invariant rules. These rules allow us to conclude that such invariants hold by proving assertions of the form $\theta[P]_x\Phi$ over trusted programs or system interfaces; note that because of the way the semantics of the modal formula is defined, the invariants hold irrespective of concurrent actions by other trusted programs and the adversary, although the assertion only refers to actions of one thread X.

Recently, Bhargavan et al. developed a type system to modularly check interfaces of security protocols, implemented the system, and applied it to analysis of secrecy properties of cryptographic protocols [27]. Their approach is based on refinement types (that is, ordinary types qualified with logical assertions), which can be used to specify program invariants and pre- and postconditions. Programmers annotate various points in the model with assumed and asserted facts. The main safety theorem states that all programmer defined assertions are implied by programmer assumed facts in a well-typed program.

However, a semantic connection between the program state and the logical formulas representing assumed and asserted facts is missing. In contrast, we prove that the inference systems of our logics of programs (PCL and LS^2) are sound with respect to trace semantics of the programming language. Our logic of programs may provide a semantic foundation for the work of Bhargavan et al. and, dually, the implementation in that work may provide a basis for

mechanizing the formal system in our logics of programs. Bhargavan et al.'s programming model is more expressive than ours because it allows higher-order functions. We intend to add higher-order functions to our framework in the near future.

While all the approaches previously discussed involve proving safety properties of protocols and systems modeled as programs, an alternative approach to secure composition involves comparing the real protocol (or system) whose security we are trying to evaluate to an ideal functionality that is secure by construction and prove that the two are equivalent in a precise sense. Once the equivalence between the real protocol and the ideal functionality is established, the composition theorem guarantees that any larger system that uses the real protocol is equivalent to the system where the real protocol is replaced by the ideal functionality.

This approach has been taken in the UC framework for cryptographic protocols [25, 26] and is also related to the notion of observational equivalence and simulation relations studied in the programming languages and verification literature [30, 31]. When possible, this form of composition result is indeed very strong: Composition is guaranteed under no assumptions about the environment in which a component is used. However, components that share state and rely on one another to satisfy certain assumptions about how that state is manipulated cannot be compositionally analyzed using this approach; the secure rely-guarantee principle we develop is better suited for such analyses. One example is the compositional security analysis of the Kerberos protocol that proceeds from proofs of its constituent programs [18].

7. Future work

There are several directions for further work on this topic. First, automating the compositional reasoning principles we presented is an open problem. Rely-guarantee reasoning principles have already been automated for functional verification of realistic systems. We expect that progress can be made on this problem by building on these prior results. Second, while sequential composition of secure systems is an important step forward, a general treatment of additive composition that considers other forms of composition is still missing. Third, it is important to extend the compositional reasoning principles presented here to support analysis of more refined models that consider, for example, features of implementation languages such as C. Finally, a quantitative theory of compositional security that supports analysis of systems built from components that are not perfectly secure would be a significant result.

About the authors

Anupam Datta is an assistant research professor at Carnegie Mellon University. Dr. Datta's research focuses on foundations of security and privacy. He has made contributions toward advancing the scientific understanding of security protocols, privacy in organizational processes, and trustworthy software systems. Dr. Datta has coauthored a book and over 30 publications in conferences and journals on these topics. He serves on the Steering Committee of the IEEE Computer Security Foundations Symposium (CSF), and has served as general chair of CSF 2008 and as program chair of the 2008 Formal and Computational Cryptography Workshop and the 2009 Asian Computing Science Conference. Dr. Datta obtained MS and PhD degrees from Stanford University and a BTech from the Indian Institute of Technology, Kharagpur, all in computer science.

John C. Mitchell is the Mary and Gordon Crary Family Professor in the Stanford Computer Science Department. His research in computer security focuses on trust management, privacy, security analysis of network protocols, and web security. He has also worked on programming language analysis and design, formal methods, and other applications of mathematical logic to computer science. Professor Mitchell is currently involved in the multiuniversity Privacy, Obligations, and Rights in Technology of Information Assessment (PORTIA) research project to study privacy concerns in databases and information processing systems, and the National Science Foundation Team for Research in Ubiquitous Secure Technology (TRUST) Center.

References

[1] Wing JM. A call to action: Look beyond the horizon. *IEEE Security & Privacy.* 2003;1(6):62–67. DOI: 10.1109/MSECP.2003.1253571

[2] Asokan N, Niemi V, Nyberg K. Man-in-the-middle in tunnelled authentication protocols. In: Christianson B, Crispo B, Malcolm JA, Roe M, editors. *Security Protocols 11th International Workshop, Cambridge, UK, April 2-4, 2003, Revised Selected Papers.* Berlin (Germany): Springer-Verlag; 2005. p. 28–41. ISBN 13: 978-354-0-28389-8

[3] Kuhlman D, Moriarty R, Braskich T, Emeott S, Tripunitara M. A correctness proof of a mesh security architecture. In: *Proceedings of the 21st IEEE Computer Security Foundations Symposium;* Jun 2008; Pittsburgh, MA. p. 315–330. DOI: 10.1109/CSF.2008.23

[4] Meadows C, Pavlovic D. Deriving, attacking and defending the GDOI protocol. In: *Proceedings of the Ninth European Symposium on Research in Computer Security;* Sep 2004; Sophia Antipolis, France. p. 53–72. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.112.3254&rep=rep1&type=pdf

[5] Mitchell JC, Shmatikov V, Stern U. Finite-state analysis of SSL 3.0. In: *Proceedings of the Seventh Conference on USENIX Security Symposium;* Jan 1998; San Antonio, TX. p. 16. Available at: http://www.usenix.org/publications/library/proceedings/sec98/mitchell.html

[6] Barth A, Jackson C, Mitchell JC. Robust defenses for cross-site request forgery. In: *Proceedings of the 15th ACM Conference on Computer and Communications Security;* Oct 2008; Alexandria, VA. p. 75–88. DOI: 10.1145/1455770.1455782

[7] Barth A, Jackson C, Mitchell JC. Securing frame communication in browsers. In: *Proceedings of the 17th USENIX Security Symposium;* Jul 2008; San Jose, CA. p. 17–30. Available at: http://www.usenix.org/events/sec08/tech/full_papers/barth/barth.pdf

[8] Chen S, Mao Z, Wang YM, Zhang M. Pretty-bad-proxy: An overlooked adversary in browsers' HTTPS deployments. In: *Proceedings of the 30th IEEE Symposium on Security and Privacy;* May 2009; Oakland, CA. p. 347–359. DOI: 10.1109/SP.2009.12

[9] Jackson C, Barth A. ForceHTTPS: Protecting high-security web sites from network attacks. In: *Proceedings of the 17th International Conference on World Wide Web;* Apr 2008; Beijing, China. p. 525–534. Available at: http://www2008.org/papers/pdf/p525-jacksonA.pdf

[10] Jackson C, Barth A, Bortz A, Shao W, Boneh D. Protecting browsers from DNS rebinding attacks. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security;* Oct 2007; Alexandria, VA. p. 421–431. DOI: 10.1145/1315245.1315298

[11] Cai X, Gui Y, Johnson R. Exploiting Unix file-system races via algorithmic complexity attacks. In: *Proceedings of the 30th IEEE Symposium on Security and Privacy;* May 2009; Oakland, CA; p. 27–41. DOI: 10.1109/SP.2009.10

[12] Datta A, Franklin J, Garg D, Kaynar D. A logic of secure systems and its application to trusted computing. In: *Proceedings of the 30th IEEE Symposium on Security and Privacy;* May 2009; Oakland, CA. p. 221–236. DOI: 10.1109/SP.2009.16

[13] Tsafrir D, Hertz T, Wagner D, Da Silva D. Portably solving file TOCTTOU races with hardness amplification. In: *Proceedings of the Sixth USENIX Conference on File and Storage Technologies;* Feb 2008; San Jose, CA. p. 1–18. Available at: http://www.usenix.org/events/fast08/tech/tsafrir.html

[14] Datta A, Derek A, Mitchell JC, Pavlovic D. A derivation system and compositional logic for security protocols. *Journal of Computer Security.* 2005;13(3):423–482. Available at: http://seclab.stanford.edu/pcl/papers/ddmp-jcs05.pdf

[15] Datta A, Derek A, Mitchell JC, Roy A. Protocol composition logic (PCL). *Electronic Notes in Theoretical Computer Science.* 2007;172:311–358. DOI: 10.1016/j.entcs.2007.02.012

[16] Durgin N, Mitchell JC, Pavlovic D. A compositional logic for proving security properties of protocols. *Journal of Computer Security.* 2003;11(4):677–721. Available at: http://www-cs-students.stanford.edu/~nad/papers/comp-jcs205.pdf

[17] Garg D, Franklin J, Kaynar DK, Datta A. Compositional system security with interface-confined adversaries. *Electronic Notes in Theoretical Computer Science.* 2010;265:49–71. DOI: 10.1016/j.entcs.2010.08.005

[18] Roy A, Datta A, Derek A, Mitchell JC, Seifert JP. Secrecy analysis in protocol composition logic. In: Okada M, Satoh I, editors. *Advances in Computer Science – ASIAN 2006: Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006.* Berlin (Germany): Springer-Verlag; 2007. p. 197–213.

[19] Butler KRB, McLaughlin SE, McDaniel PD. Kells: A protection framework for portable data. In: *Proceedings of the 26th Annual Computer Security Applications Conference;* Dec 2010; Austin, TX. p. 231–240. DOI: 10.1145/1920261.1920296

[20] Kannan J, Maniatis P, Chun B. Secure data preservers for web services. In: *Proceedings of the Second USENIX Conference on Web Application Development;* Jun 2011; Portland, OR. p. 25–36. Available at: http://www.usenix.org/events/webapps11/tech/final_files/Kannan.pdf

[21] He C, Sundararajan M, Datta A, Derek A, Mitchell JC. A modular correctness proof of IEEE 802.11i and TLS. In: *Proceedings of the 12th ACM Conference on Computer and Communications Security;* Nov 2005; Alexandria, VA. p. 2–15. DOI: 10.1145/1102120.1102124

[22] Datta A, Derek A, Mitchell JC, Pavlovic D. Abstraction and refinement in protocol derivation. In: *Proceedings of 17th IEEE Computer Security Foundations Workshop;* Jun 2004; Pacific Grove, CA. p. 30–45. DOI: 10.1109/CSFW.2004.1310730

[23] Jones CB. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems.* 1983;5(4):596–619. DOI: 10.1145/69575.69577

[24] Misra J, Chandy KM. Proofs of networks of processes. *IEEE Transactions on Software Engineering.* 1981;7(4):417–426. DOI: 10.1109/TSE.1981.230844

[25] Canetti R. Universally composable security: A new paradigm for cryptographic protocols. In: *Proceedings of the 42nd IEEE Symposium on the Foundations of Computer Science;* Oct 2001; Las Vegas, NV. p. 136–145. DOI: 10.1109/SFCS.2001.959888

[26] Pfitzmann B, Waidner M. A model for asynchronous reactive systems and its application to secure message transmission. In: *IEEE Symposium on Security and Privacy;* May 2001; Oakland, CA. p. 184–200. DOI: 10.1109/SECPRI.2001.924298

[27] Bhargavan K, Fournet C, Gordon AD. Modular verification of security protocol code by typing. In: *Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages;* Jan 2010; Madrid, Spain. p. 445–456. DOI: 10.1145/1706299.1706350

[28] Paulson L. Proving properties of security protocols by induction. In: *Proceedings of 10th IEEE Computer Security Foundations Workshop;* Jun 1997; Rockport, MA. p. 70–83. DOI: 10.1109/CSFW.1997.596788

[29] Alpern B, Schneider FB. Recognizing safety and liveness. *Distributed Computing.* 1987;2(3):117–126. DOI: 10.1007/BF01782772

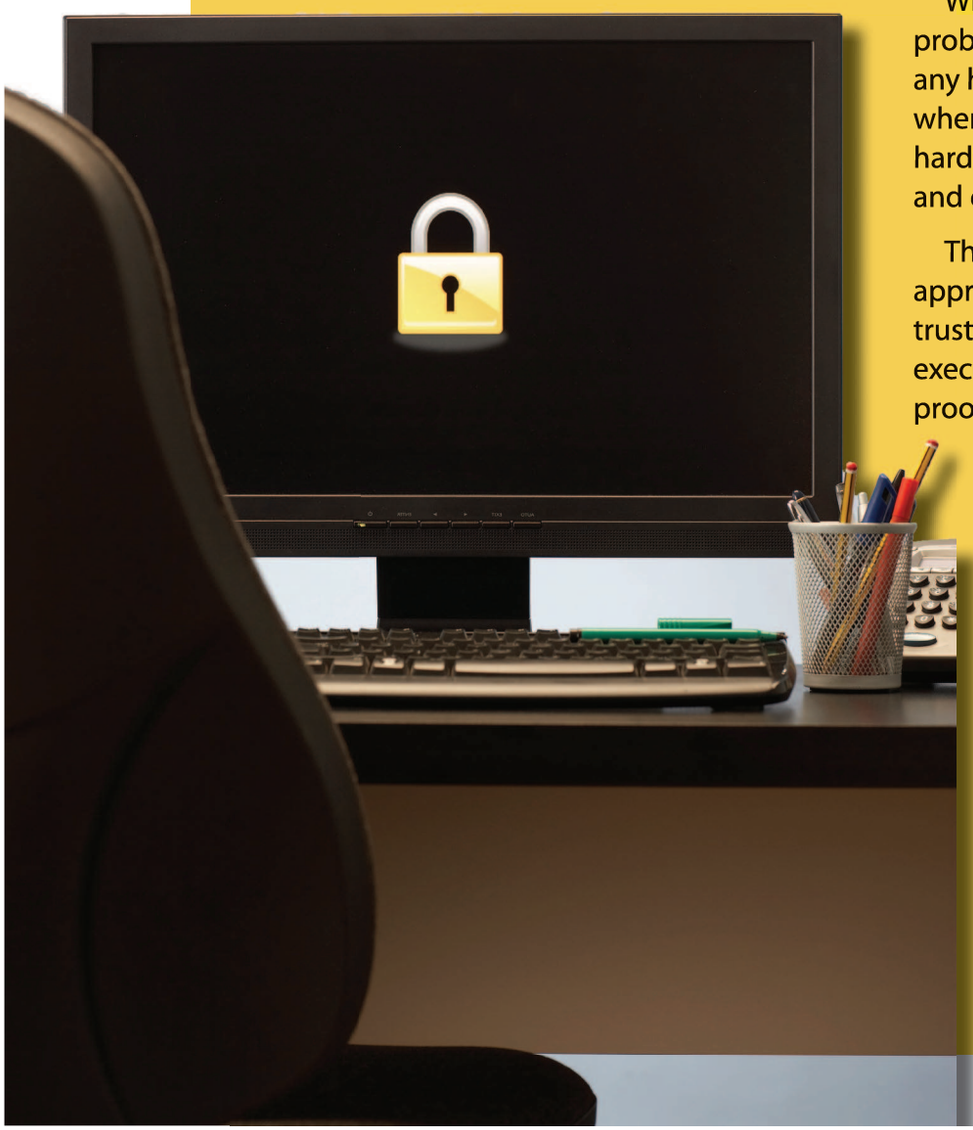
[30] Canetti R, Cheung L, Kaynar DK, Liskov M, Lynch NA, Pereira O, Segala R. Time-bounded task-PIOAs: A framework for analyzing security protocols. In: *Proceedings of the 20th International Symposium on Distributed Computing;* Sep 2006; Stockholm, Sweden. p. 238–253. DOI: 10.1007/11864219_17

[31] Küsters R, Datta A, Mitchell JC, Ramanathan A. On the relationships between notions of simulation-based security. *Journal of Cryptology.* 2008;21(4):492–546. DOI: 10.1007/s00145-008-9019-9

Proof-carrying data: Secure computation on untrusted platforms

Alessandro Chiesa and Eran Tromer

When running software applications and services, we rely on the underlying execution platform: the hardware and the lower levels of the software stack. The execution platform is susceptible to a wide range of threats, ranging from accidental bugs, faults, and leaks to maliciously induced Trojan horses. The problem is aggravated by growing system complexity and by increasingly pertinent outsourcing and supply chain consideration. Traditional mechanisms, which painstakingly validate all system components, are expensive and limited in applicability.



What if the platform assurance problem is just too hard? Do we have any hope of securely running software when we cannot trust the underlying hardware, hypervisor, kernel, libraries, and compilers?

This article will discuss a potential approach for doing just so: conducting trustworthy computation on untrusted execution platforms. The approach, proof-carrying data (PCD), circumnavigates the threat of faults and leakage by reasoning solely about properties of a computation's output data, regardless of the process that produced it. In PCD, the system designer prescribes the desired properties of the computation's outputs. These properties are then enforced using cryptographic proofs attached to all data flowing through the system and verified at the system perimeter as well as internal nodes.

1. Introduction

Integrity of data, information flow control, and fault isolation are three examples of security properties of which attainment, in the general case and under minimal assumptions, is a major open problem. Even when particular solutions for specific cases are known, they tend to rely on platform trust assumptions (for example, the kernel is trusted, the central processing unit is trusted), and even then they cannot cross trust boundaries between mutually untrusting parties. For example, in cloud computing, clients are typically interested in both integrity [1] and confidentiality [2] when they delegate their own computations to the untrusted workers.

Minimal trust assumptions and very strong certification guarantees are sometimes almost a basic requirement. For example, within the information technology supply chain, faults can be devastating to security [3] and hard to detect; moreover, hardware and software components are often produced in faraway lands from parts of uncertain origin where it is hard to carry out quality assurance in case trust is not available [4]. This all implies risks to the users and organizations [5, 6, 7, 8].

2. Goals

In order to address the aforementioned problems, we propose the following goal:

> **Goal.** A compiler that, given a protocol for a distributed computation and a security property (in the form of a predicate to be verified at every node of the computation), yields an augmented protocol that enforces the security property.

We wish this compiler to *respect the original distributed computation* (that is, the compiler should preserve the computation's communication graph, dynamics, and efficiency). This implies, for example, that *scalability* is preserved: If the original computation can be jointly conducted by numerous parties, then the compiler produces a secure distributed computation that has the same property.

3. Our approach

We propose a generic solution approach, proof-carrying data (PCD), to solve the aforementioned problems by defining appropriate checks to be performed on each party's computation and then letting parties attach proofs of correctness to each message. Every piece of data flowing through a distributed computation is augmented by a short proof string that certifies the data as compliant with some desired property. These proofs can be propagated and aggregated "on the fly," as the computation proceeds. These proofs may be between components of a single platform or between components of mutually untrusting platforms, thereby extending trust to any distributed computation.

But what "properties" do we consider? Certainly we want to consider the property that every node carried out its own computation without making any mistakes. More generally, we consider properties that can be expressed as a requirement that every step in the computation satisfies some *compliance predicate* C computable in polynomial time; we call this notion *C-compliance*. Thus, each party receives inputs that are augmented with proof strings, computes some outputs, and augments each of the outputs with a new proof string that will convince the next party (or the verifier of the ultimate output) that the output is consistent with a C-compliant computation. See figure 1 for a high-level diagram of this idea.

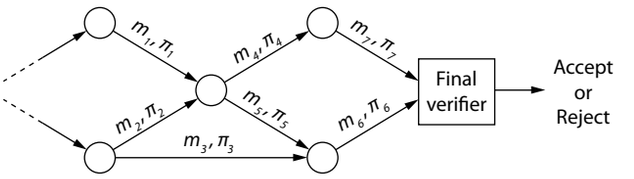


FIGURE 1. A distributed computation in which each party sends a message m_i that is augmented with a short proof π_i. The final verifier inspects the computation's outputs in order to decide whether they are "compliant" or not.

For example, C could simply require that each party's computation was carried out without errors. Or, C could require that not only each party's computation was carried out without errors, but also that the program run by each party carried a signature valid under the system administrator's public key; in such a case, the *local program* supplied by each party would be the combination of the program and the signature. Or, C could alternatively require that each party's computation involved a binary produced by

a compiler prescribed by the system administrator, which is known to perform certain tests on the code to be compiled (for example, type safety, static analysis, dynamic enforcement). Note that a party's local program could be a combination of code, human inputs, and randomness.

To formalize the above, we define and construct a PCD scheme: A cryptographic primitive that fully encapsulates the proof system machinery and provides a simple but very general "interface" to be used in applications.[a]

Our construction does require a minimal trusted setup: Every party should have black-box access to a simple *signed-input-and-randomness* functionality, which signs every input it receives along with some freshly-generated random bits. This is similar to standard functionality of cryptographic signing tokens and can also be implemented using Trusted Platform Module chips or a trusted party.

3.1. Our results

We introduce the generic approach of PCD for securing distributed computations and describing the cryptographic primitive of PCD schemes to capture this approach:

> **Theorem (informal).** PCD schemes can be constructed under standard cryptographic assumptions, given signed-input-and-randomness tokens.

3.2. The construction and its practicality

We do not rely on the traditional notion of a proof; instead, we rely on *computationally sound proofs*. These are proofs that always exist for true theorems and can be found efficiently given the appropriate witness. For false theorems, however, we only have the guarantee that no *efficient* procedure will be able to write a proof that makes us accept with more than negligible probability. Nonetheless, computationally sound proofs are just as good as traditional ones, for we are not interested in being protected against infeasible attack procedures, nor do we mind accepting a false theorem with, say, 2^{-100} probability.

The advantage of settling for computationally sound proofs is that they can be much shorter than the computation to which they attest and can be verified much more quickly than repeating the entire computation. To this end, we use probabilistically checkable proofs (PCPs) [11, 12], which originate in the field of computational complexity and its cryptographic extensions [9, 13, 14].

While our initial results establish theoretical foundations for PCD and show their possibility in principle, the aforementioned PCPs are computationally heavy and are notorious for being efficient only in the asymptotic sense, and they are not yet of practical relevance. Motivated by the potential impact of a practical PCD scheme, we have thus taken on the challenge of constructing a practical PCP system, in an ongoing collaboration with Professor Eli Ben-Sasson and a team of programmers at the Technion.

4. Related approaches

Cryptographic tools. Secure multiparty computation [15, 16, 17] considers the problem of secure function evaluation; our setting is not *one* function evaluation, but ensuring a single invariant (that is, *C*-compliance) through *many* interactions and computations between parties.

Platforms, languages, and static analysis. Integrity can be achieved by running on suitable fault-tolerant systems. Confidentiality can be achieved by platforms with suitable information flow control mechanisms following [18, 19] (for example, at the operating-system level [20, 21]). Various invariants can be achieved by statically analyzing programs and by programming language mechanisms such as type systems following [22, 23]. The inherent limitation of these approaches is that the output of such computation can be trusted only if one trusts the whole platform that executed it; this renders them ineffective in the setting of mutually untrusting distributed parties.

Run-time approaches. In proof-carrying code (PCC) [24], the code producer augments the code with formal, efficiently checkable proofs of the desired properties (typically, using the aforementioned language or static analysis techniques); PCC and PCD are

a. PCD schemes generalize the "computationally-sound proofs" of Micali [9], which consider only the "one-hop" case of a single prover and a single verifier and also generalize the "incrementally verifiable computation" of Valiant [10], which considers the case of an a-priori fixed sequence of computations.

complementary techniques, in the sense that PCD can enforce properties expressed via PCC. Dynamic analysis monitors the properties of a program's execution at run-time (for example, [25, 26, 27]). Our approach can be interpreted as extending dynamic analysis to the distributed setting, by allowing parties to (implicitly) monitor the program execution of all prior parties without actually being present during the executions. The Fabric system [28] is similar to PCD in motivation, but takes a very different approach: Fabric aims to make maximal use of distributed-system *given* trust constraints, while PCD *creates* new trust relations.

5. The road onward

We envision PCD as a framework for achieving security properties in a nonconventional way that circumvents many difficulties with current approaches. In PCD, faults and leakage are acknowledged as an expected occurrence, and rendered inconsequential by reasoning about properties of data that are independent of the preceding *computation.* The system designer prescribes the desired properties of the computation's output; proofs of these properties are attached to the data flowing through the system and are mutually verified by the system's components.

We have already shown explicit constructions of PCD, under standard cryptographic assumptions, in the model where parties have black-box access to a simple hardware token. The theoretical problem of weakening this requirement, or formally proving that it is (in some sense) necessary, remains open. In recent work, we show how to resolve this problem in the case of a single party's computation [29].

As for practical realizations, since there is evidence that the use of PCPs for achieving short proofs is inherent [30], we are tackling head-on the challenge of making PCPs practical. We are also studying devising ways to express the security properties, to be enforced by PCD, using practical programming languages such as C++.

In light of these, as real-world practicality of PCD becomes closer and closer, the task of *compliance engineering* becomes an exciting direction. While PCD provides a protocol compiler to ensure any compliance predicate in a distributed computation, figuring out what are useful compliance predicates in this or that setting is a problem in its own right.

We already envision problem domains where we believe enforcing compliance predicates will come a long way toward securing distributed systems in a strong sense:

- **Multilevel security.** PCD may be used for information flow control. For example, consider enforcing multilevel security [31, Chap. 8.6] in a room full of data-processing machines. We want to publish outputs labeled "nonsecret," but are concerned that they may have been tainted by "secret" information (for example, due to bugs, via software side channel attacks [32] or, perhaps, via literal eavesdropping [33, 34, 35]). PCD then allows you to reduce the problem of controlling information flow to the problem of controlling the perimeter of the information room by ensuring that every network packet leaving the room is inspected by the PCD verifier to establish it carries a valid proof.
- **IT supply chain and hardware Trojans.** Using PCD, one can achieve fault isolation and accountability at the level of system components (for example, chips or software modules) by having each component augment every output with a proof that its computation, *including all history it relied on,* was correct. Any fault in the computation, malicious or otherwise, will then be identified by the first nonfaulty subsequent component. Note that even the PCD verifiers themselves do not have to be trusted except for the very last one.
- **Distributed type safety.** Language-based type-safety mechanisms have tremendous expressive power, but are targeted at the case where the underlying execution platform can be trusted to enforce type rules. Thus, they typically cannot be applied across distributed systems consisting of multiple mutually untrusting execution platforms. This barrier can be surmounted by using PCD to augment typed values passing between systems with proofs for the correctness of the type.

Efforts to understand how to think about compliance in concrete problem domains are likely to uncover common problems and corresponding design patterns [36], thus improving our overall ability to correctly phrase desired security properties as compliance predicates.

We thus pose the following challenge: Given a genie that grants every wish expressed as a compliance predicate on distributed computations, what compliance predicates would you wish for in order to achieve the security properties your system needs?

Acknowledgments

This research was partially supported by the Check Point Institute for Information Security, the Israeli Centers of Research Excellence program (center No. 4/11), the European Community's Seventh Framework Programme grant 240258, the National Science Foundation (NSF) grant NSF-CNS-0808907, and the Air Force Research Laboratory (AFRL) grant FA8750-08-1-0088. Views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AFRL, NSF, the US government or any of its agencies.

About the authors

Alessandro Chiesa is a second-year doctoral student in the Theory of Computation group in the Computer Science and Artificial Intelligence Laboratory (CSAIL) at Massachusetts Institute of Technology (MIT). He is interested in cryptography, complexity theory, quantum computation, mechanism design, algorithms, and security. He can be reached at MIT CSAIL, alexch@csail.mit.edu.

Eran Tromer is a faculty member at the School of Computer Science at Tel Aviv University. His research focus is information security, cryptography, and algorithms. He is particularly interested in what happens when cryptographic systems meet the real world, where computation is faulty and leaky. He can be reached at Tel Aviv University, tromer@cs.tau.ac.il.

References

[1] Ferdowsi A. S3 data corruption? *Amazon Web Services* (discussion forum). 2008 Jun 22. Available at: https://forums.aws.amazon.com/thread.jspa?threadID=22709&start=0&tstart=0

[2] Ristenpart T, Tromer E, Shacham H, Savage S. Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds. In: *Proceedings of the 16th ACM Conference on Computer and Communications Security;* Nov 2009; Chicago, IL. p. 199–212. Available at: http://cseweb.ucsd.edu/~hovav/dist/cloudsec.pdf

[3] Biham E, Shamir A. Differential fault analysis of secret key cryptosystems. In: Kaliski BS Jr., editor. *Advances in Cryptology—CRYPTO '97* (Proceedings of the 17th Annual International Cryptology Conference; Aug 1997; Santa Barbara, CA). LNCS, 1294. London (UK): Springer-Verlag; 1997. p. 513–525. DOI: 10.1007/BFb0052259

[4] Collins DR. Trust, a proposed plan for trusted integrated circuits. Paper presented at a conference; Mar 2006; p. 276–277. Available at: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA456459

[5] Agrawal D, Baktir S, Karakoyunlu D, Rohatgi P, Sunar B. Trojan detection using IC fingerprinting. In: *Proceedings of the 2007 IEEE Symposium on Security and Privacy;* May 2007; Oakland, CA. p. 296–310. DOI: 10.1109/SP.2007.36

[6] Biham E, Carmeli Y, Shamir A. Bug attacks. In: Wagner D, editor. *Advances in Cryptology—CRYPTO 2008* (Proceedings of the 28th Annual International Cryptology Conference; Aug 2008; Santa Barbara, CA). LNCS, 5157. Berlin (Germany): Springer-Verlag; 2008. p. 221–240. DOI: 10.1007/978-3-540-85174-5_13

[7] King ST, Tucek J, Cozzie A, Grier C, Jiang W, Zhou Y. Designing and implementing malicious hardware. In: *Proceedings of the First USENIX Workshop on Large-Scale Exploits and Emergent Threats;* Apr 2008; San Francisco, CA. p. 1–8. Available at: http://www.usenix.org/events/leet08/tech/full_papers/king/king.pdf

[8] Roy JA, Koushanfar F, Markov IL. Circuit CAD tools as a security threat. In: *Proceedings of the First IEEE International Workshop on Hardware-Oriented Security and Trust;* Jun 2008; Anaheim, CA. p. 65–66. DOI: 10.1109/HST.2008.4559052

[9] Micali S. Computationally sound proofs. *SIAM Journal on Computing.* 2000;30(4):1253–1298. DOI: 10.1137/S0097539795284959

[10] Valiant P. Incrementally verifiable computation or

proofs of knowledge imply time/space efficiency. In: Canetti R, editor. *Theory of Cryptography* (Proceedings of the Fifth Theory of Cryptography Conference; Mar 2008; New York, NY). LNCS, 4948. Berlin (Germany): Springer-Verlag; 2008. p. 1–18. DOI: 10.1007/978-3-540-78524-8_1

[11] Babai L, Fortnow L, Levin LA, Szegedy M. Checking computations in polylogarithmic time. In: *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing;* May 1991; New Orleans, LA. p. 21–32. DOI: 10.1145/103418.103428

[12] Ben-Sasson E, Sudan M. Simple PCPs with poly-log rate and query complexity. In: *Proceedings of the 37th Annual ACM Symposium on Theory of Computing;* May 2005; Baltimore, MD. p. 266–275. DOI: 10.1145/1060590.1060631

[13] Kilian J. A note on efficient zero-knowledge proofs and arguments. In: *Proceedings of the 24th Annual ACM Symposium on Theory of Computing;* May 1992; Victoria, BC, Canada. p. 723–732. DOI: 10.1145/129712.129782

[14] Barak B, Goldreich O. Universal arguments and their applications. In: *Proceedings of the 17th IEEE Annual Conference on Computational Complexity;* May 2002; Montreal, Quebec , Canada. p. 194–203. DOI: 10.1109/CCC.2002.1004355

[15] Goldreich O, Micali S, Wigderson A. How to play ANY mental game. In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing;* May 1987; New York, NY. p. 218–229. DOI: 10.1145/28395.28420

[16] Ben-Or M, Goldwasser S, Wigderson A. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing;* May 1988; Chicago, IL. p. 1–10. DOI: 10.1145/62212.62213

[17] Chaum D, Crépeau C, Damgård I. Multiparty unconditionally secure protocols. In: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing;* May 1988; Chicago, IL. p. 11–19. DOI: 10.1145/62212.62214

[18] Denning DE, Denning PJ. Certification of programs for secure information flow. *Communications of the ACM.* 1977;20(7):504–513. DOI: 10.1145/359636.359712

[19] Myers AC, Liskov B. A decentralized model for information flow control. In: *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles;* Oct 1997; Saint-Malo, France. p. 129–142. DOI: 10.1145/268998.266669

[20] Krohn M, Yip A, Brodsky M, Cliffer N, Kaashoek MF, Kohler E, Morris R. Information flow control for standard OS abstractions. In: *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles;* Oct 2007; Stevenson, WA. p. 321–334. DOI: 10.1145/1294261.1294293

[21] Zeldovich N, Boyd-Wickizer S, Kohler E, Mazières D. Making information flow explicit in HiStar. In: *Proceedings of the Seventh USENIX Symposium on Operating Systems Design and Implementation;* Nov 2006; Seattle, WA. p. 19–19. Available at: http://www.usenix.org/event/osdi06/tech/full_papers/zeldovich/zeldovich.pdf

[22] Andrews GR, Reitman RP. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems.* 1980;2(1):56–76. DOI: 10.1145/357084.357088

[23] Denning DE. A lattice model of secure information flow. *Communications of the ACM.* 1976;19(5):236–243. DOI: 10.1145/360051.360056

[24] Necula GC. Proof-carrying code. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages;* Jan 1997; Paris, France. p. 106–119. DOI: 10.1145/263699.263712

[25] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation;* Jun 2007; San Diego, CA. p. 89–100. DOI: 10.1145/1250734.1250746

[26] Suh GE, Lee JW, Zhang D, Devadas S. Secure program execution via dynamic information flow tracking. In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems;* Oct 2004; Boston, MA. p. 85–96. DOI: 10.1145/1024393.1024404

[27] Kiriansky V, Bruening D, Amarasinghe SP. Secure execution via program shepherding. In: *Proceedings of the 11th USENIX Security Symposium;* Aug 2002; San Francisco, CA. p. 191–206. Available at: http://www.usenix.org/publications/library/proceedings/sec02/full_papers/kiriansky/kiriansky_html/index.html

[28] Liu J, George MD, Vikram K, Qi X, Waye L, Myers AC. Fabric: A platform for secure distributed computation and storage. In: *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles;* Oct 2009; Big Sky, MT. p. 321–334. DOI: 10.1145/1629575.1629606

[29] Bitansky N, Canetti R, Chiesa A, Tromer E. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. Cryptology ePrint Archive. 2011;Report 2011/443. Available at: http://eprint.iacr.org/2011/443

[30] Rothblum GN, Vadhan S. Are PCPs inherent in efficient arguments? In: *Proceedings of the 24th IEEE Annual Conference on Computational Complexity;* Jul 2009; Paris, France. p. 81–92. DOI: 10.1109/CCC.2009.40

[31] Anderson RJ. *Security Engineering: A Guide to Building Dependable Distributed Systems*. 2nd ed. Indianapolis (IN): Wiley Publishing; 2008. ISBN: 978-0-470-06852-6

[32] Brumley D, Boneh D. Remote timing attacks are practical. *Computer Networks: The International Journal of Computer and Telecommunications Networking.* 2005;48(5):701–716.

[33] LeMay M, Tan J. Acoustic surveillance of physically unmodified PCs. In: *Proceedings of the 2006 International Conference on Security and Management;* Jun 2006; Las Vegas, NV. p. 328–334. Available at: http://ww1.ucmss.com/books/LFS/CSREA2006/SAM4311.pdf

[34] Asonov D, Agrawal R. Keyboard acoustic emanations. In: *Proceedings of the 2004 IEEE Symposium on Security and Privacy;* May 2004; Oakland, CA. p. 3–11. DOI: 10.1109/SECPRI.2004.1301311

[35] Tromer E, Shamir A. Acoustic cryptanalysis: On nosy people and noisy machines. Presentation at: *Eurocrypt 2004 Rump Session;* May 2004; Interlaken, Switzerland. Available at: http://people.csail.mit.edu/tromer/acoustic

[36] Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston (MA): Addison-Wesley Longman Publishing Co., Inc.; 1995. ISBN: 9780201633610

Blueprint for a science of cybersecurity

Fred B. Schneider

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

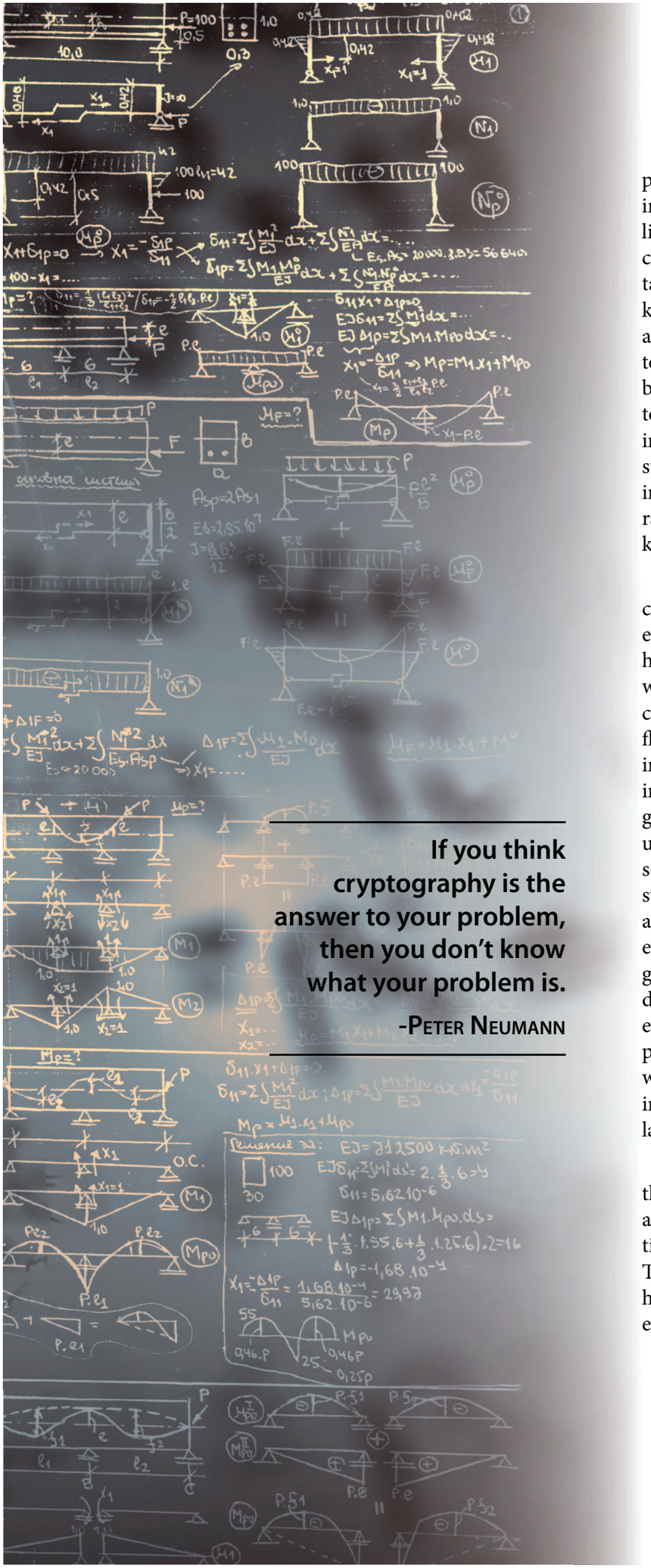
- transcend specific technologies and attacks, yet still be applicable in real settings,
- introduce new models and abstractions, thereby bringing pedagogical value besides predictive power, and
- facilitate discovery of new defenses as well as describe non-obvious connections between attacks, defenses, 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.

BOX 1. What is a science?

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.



If you think cryptography is the answer to your problem, then you don't know what your problem is.

-Peter Neumann

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 [1] 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 someday 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 coevolve 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 article 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 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, then, are crucial 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. [2] 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 [3]. (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

FIGURE 1. Phishing attacks, which enable theft of passwords and ultimately facilitate identity theft, can be classified according to how the human user is fooled into empowering the adversary.

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** refers to which principals are allowed to learn what information.
- **Integrity** refers to what changes to the system (stored information and resource usage) and to its environment (outputs) are allowed.
- **Availability** refers to 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 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.[a] 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.

Contrast this state of affairs with trace properties, where safety ("no 'bad thing' happens") and liveness ("some 'good thing' happens") are orthogonal classes. (Formal definitions of trace properties, safety, and liveness are given in box 2 for those readers who are interested.) 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 [5], 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.

BOX 2. Trace properties, safety, and liveness

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 [6] 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.

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

Every trace property is either safety, liveness, or the conjunction of two trace properties—one that is safety and one that is liveness [7]. 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 [8]. Thus, the safety-liveness classification for trace properties comes with proof methods beyond offering formal definitions.

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 indivisible 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 operating system 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 type-checking, 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.

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, operating system 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

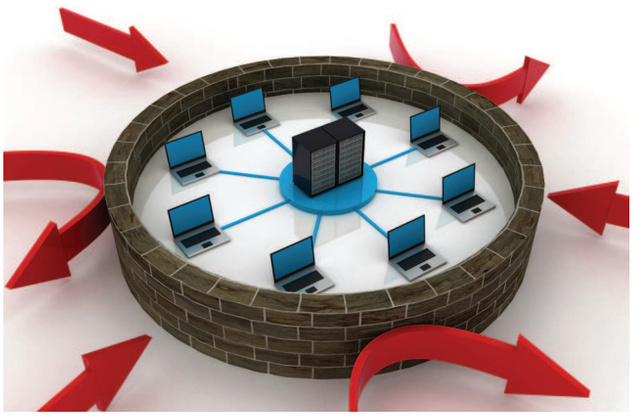


FIGURE 2. A firewall is an example of a reference monitor.

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 coevolve 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. Next, we discuss a law [9] 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 [10]. For every trace property ψ that is safety, there exists an automaton m_ψ that accepts the set of traces in ψ [8].

Automaton m_ψ is a reference 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 3, 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

$$\textbf{if } M_\psi(\text{“}S_i\text{”}) \neq \text{OK } \textbf{then halt} \qquad (1)$$

in figure 3 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.

S_1		**if** M_ψ("S_1") ≠ OK **then halt**
S_2		S_1
S_3	⇒	**if** M_ψ("S_2") ≠ OK **then halt**
S_4		S_2
…		…
original		**inlined reference monitor**

FIGURE 3. Inlined reference monitor example

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.[b] 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 τ_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 formulating 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

b. There is also experimental evidence [11] that distinct versions built by independent teams nevertheless share vulnerabilities.

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.[c] 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 program satisfies a specification and for justifying stepwise 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.

BOX 3. Satisfies and refinement

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 stepwise 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 stepwise refinement is not sound for deriving programs that satisfy this policy.

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* [13], 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

c. 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).

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 interconnections 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 sciences is supported by laboratories, experimentation for a science of cybersecurity will require test beds 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 a workshop organized by the National Science Foundation (NSF), NSA, and the Intelligence Advanced Research Projects Activity 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 Team for Research in Ubiquitous Secure Technology (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.

Funding

This research is supported in part by NSF grants 0430161, 0964409, and CCF-0424422 (TRUST), Office of Naval Research 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 US Government.

About the author

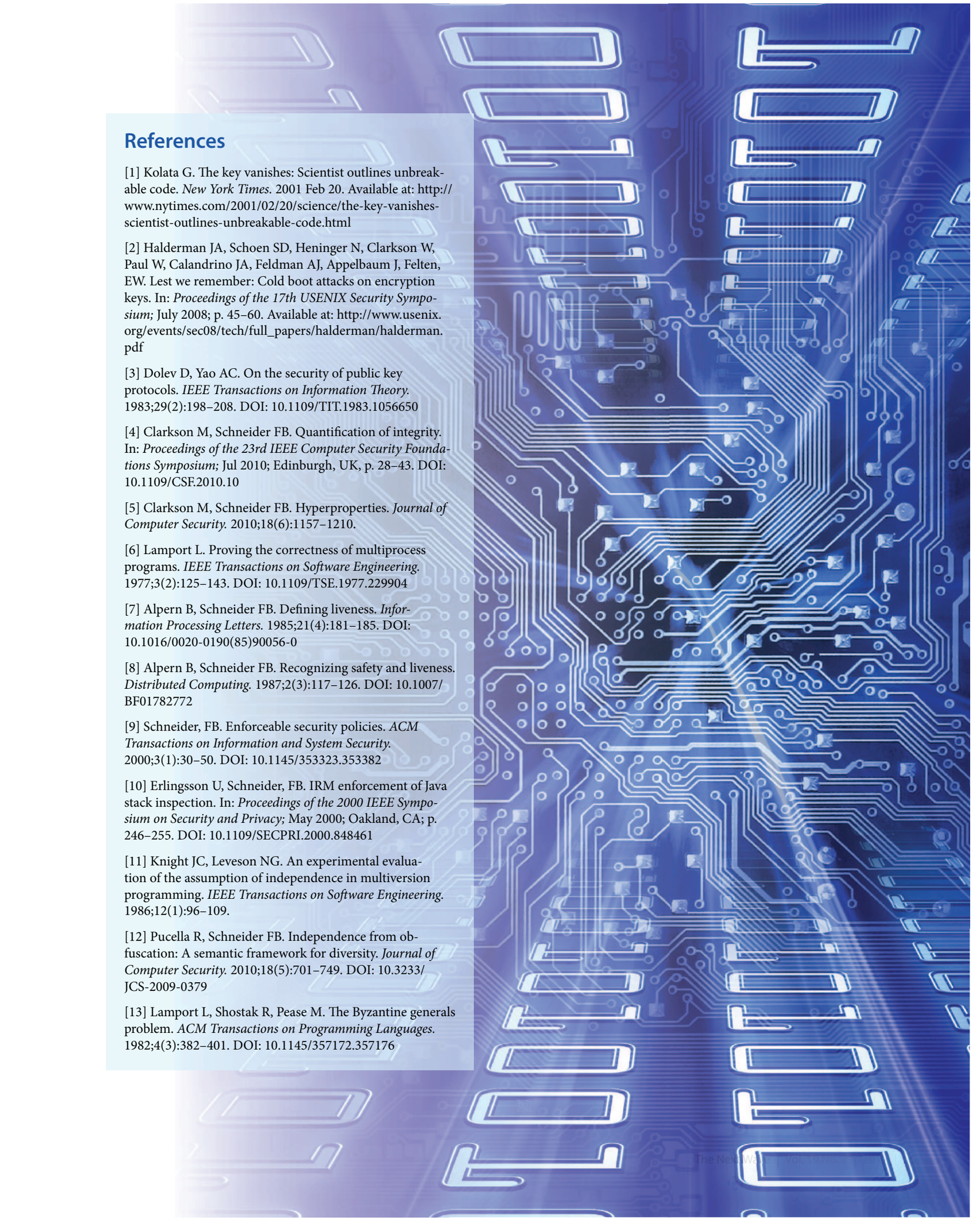
Fred B. Schneider joined the Cornell University faculty in 1978, where he is now the Samuel B. Eckert Professor of Computer Science. He also is the chief scientist of the NSF TRUST Science and Technology Center, and he has been professor at large at the University of Tromso since 1996. He received a BS from Cornell University (1975) and a PhD from Stony Brook University (1978).

Schneider's research concerns trustworthy systems, most recently focusing on computer security. His early work was in formal methods and fault-tolerant distributed systems. He is author of the graduate textbook *On Concurrent Programming,* coauthor (with David Gries) of the undergraduate text *A Logical Approach to Discrete Math,* and the editor of *Trust in Cyberspace,* which reports findings from the US National Research Council's study that Schneider chaired on information systems trustworthiness.

A fellow of the American Association for the Advancement of Science, the Association for Computing Machinery, and the Institute of Electrical and Electronics Engineers, Schneider was granted a DSc honoris causa by the University of Newcastle-upon-Tyne in 2003. He was awarded membership in Norges Tekniske Vitenskapsakademi (the Norwegian Academy of Technological Sciences) in 2010 and the US National Academy of Engineering in 2011. His survey paper on state machine replication received a Special Interest Group on Operating Systems (SIGOPS) Hall of Fame Award.

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 sciences. He is also a member of the Defense Science Board and the National Institute for Standards and Technology Information Security and Privacy Advisory Board. A frequent consultant to industry, Schneider co-chairs Microsoft's Trustworthy Computing Academic Advisory Board.

Dr. Schneider can be reached at the Department of Computer Science at Cornell University in Ithaca, New York 14853.



References

[1] Kolata G. The key vanishes: Scientist outlines unbreakable code. *New York Times*. 2001 Feb 20. Available at: http://www.nytimes.com/2001/02/20/science/the-key-vanishes-scientist-outlines-unbreakable-code.html

[2] Halderman JA, Schoen SD, Heninger N, Clarkson W, Paul W, Calandrino JA, Feldman AJ, Appelbaum J, Felten, EW. Lest we remember: Cold boot attacks on encryption keys. In: *Proceedings of the 17th USENIX Security Symposium;* July 2008; p. 45–60. Available at: http://www.usenix.org/events/sec08/tech/full_papers/halderman/halderman.pdf

[3] Dolev D, Yao AC. On the security of public key protocols. *IEEE Transactions on Information Theory.* 1983;29(2):198–208. DOI: 10.1109/TIT.1983.1056650

[4] Clarkson M, Schneider FB. Quantification of integrity. In: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium;* Jul 2010; Edinburgh, UK, p. 28–43. DOI: 10.1109/CSF.2010.10

[5] Clarkson M, Schneider FB. Hyperproperties. *Journal of Computer Security.* 2010;18(6):1157–1210.

[6] Lamport L. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering.* 1977;3(2):125–143. DOI: 10.1109/TSE.1977.229904

[7] Alpern B, Schneider FB. Defining liveness. *Information Processing Letters.* 1985;21(4):181–185. DOI: 10.1016/0020-0190(85)90056-0

[8] Alpern B, Schneider FB. Recognizing safety and liveness. *Distributed Computing.* 1987;2(3):117–126. DOI: 10.1007/BF01782772

[9] Schneider, FB. Enforceable security policies. *ACM Transactions on Information and System Security.* 2000;3(1):30–50. DOI: 10.1145/353323.353382

[10] Erlingsson U, Schneider, FB. IRM enforcement of Java stack inspection. In: *Proceedings of the 2000 IEEE Symposium on Security and Privacy;* May 2000; Oakland, CA; p. 246–255. DOI: 10.1109/SECPRI.2000.848461

[11] Knight JC, Leveson NG. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering.* 1986;12(1):96–109.

[12] Pucella R, Schneider FB. Independence from obfuscation: A semantic framework for diversity. *Journal of Computer Security.* 2010;18(5):701–749. DOI: 10.3233/JCS-2009-0379

[13] Lamport L, Shostak R, Pease M. The Byzantine generals problem. *ACM Transactions on Programming Languages.* 1982;4(3):382–401. DOI: 10.1145/357172.357176