Turing is from Mars, Shannon is from Venus:

Computer Science and Computer Engineering

S.W. Smith
Dartmouth College

The aim of this department is to look at security issues in the context of larger systems. When thinking about systems, it's tempting to only envision computational elements such as machines, operating systems (OSs), and programming languages, or human elements such as user interfaces, business practices, and public policy. However, to mangle an analogy from physics, the observer is also part of the system. When reasoning about or designing (or breaking into) secure systems, it's important to remember the tools, mindset, and background we bring to the table.

Computer security's primary background fields are computer science and computer engineering (although some might make a case for mathematical logic). These fields sometimes bring very different approaches to the same basic security problems. In this installment, we take a lighthearted look at these differences.

Within universities, distinguishing the computer science department from the computer engineering department can be tricky. At my undergraduate institution, the two programs came from the same ancestor, electrical engineering and computer science. At Dartmouth College, however, computer science emerged from the mathematics department. Engineering is in a completely different school. As a consequence, although computer science and computer engineering are natural partners for computer security courses and grants, we must move up several levels in the hierarchy of deans before we find one in common, thus complicating cooperation. Nevertheless, most courses are cross-listed, leading many undergraduates starting my operating systems class to think they're taking an engineering course. (In fact, a "computer science" major is often a liberal arts program that lets students take courses in less technical subjects such as literature and history. Each camp seems to regard this difference as a weakness of the other camp.)

However murky the organizational roots, each discipline takes its own distinctive approach to computer and computation problems. At a very coarse (and, hence, wrong) granularity, computer science looks at computation as an abstract, almost mathematical process, drawing from thinkers such as Alan Turing on the fundamental nature of computation; computer engineering considers the physical systems that realize this process, holding up Claude Shannon's work on the fundamental nature of information. These roots can lead to different perspectives on security problems.

Computability theory

First, we might ask the basic question: what does computation do?

Computer scientists, drawing on Turing, think of a *function* as a map that takes each element of an input set to some output value. Because we're talking about computers, we can assume that the input set consists of strings of bits, and that the output values are 1 or 0. (Although this assumption might sound limiting, it's not—we can still express any reasonable problem this way.) The Church-Turing thesis characterizes *computing* as running a program on a device such as a modern computer, with the minor difference that the thesis assumes the computer might have unbounded memory. For some simple functions (such as "print a 1 if the input has an odd number of ones, and 0 otherwise"), it's pretty clear how we can write a program to compute them. We might naturally assume that we can write a program to compute any function.

A classic (and surprising) result of theoretical computer science is that this assumption is wrong! Functions exist that aren't computable. The way to construct troublesome examples is to realize that programs themselves can be represented as bitstrings—and then to start asking about programs that compute functions about programs. The standard example is the *halting problem*—that is, will a given program on a given input actually

halt, or will it run forever? Fundamental work in computational theory shows that we can't write a program that will get this problem right on every possible input.

Freshly armed with a PhD in computer science (and looking at the world through Turing-colored lenses) I set out as a post-doc doing security consulting at Los Alamos National Laboratory, in the early days of the World Wide Web. A large government agency approached us with a problem. They planned to construct what they called a *subweb*; users would enter the agency's site via an official entrance page, but would then wander through a large collection of pages, hosted by many different third parties (typically universities), but branded with the agency's name. The agency was worried that bored system administrators might install "Easter egg" functionality that would entertain them and their friends, but would embarrass the agency and damage its reputation. For example, suppose a Web form asked for a zip code and fed this back to a complex server-side program that produced some response to send back to the user. The agency worried that a user entering a certain zip code (say, 12345) would get a response that took the user to someplace embarrassing (say, playboy.com).

The agency asked us if we could write a Web spider that would crawl through their subweb and determine if such functionality existed. We could have certainly written a basic crawler, but these sites might have had arbitrary programs on the back end that would take Web form requests and such as input and return HTML as output.

Suppose we could write an analyzer program A that would take such a program P, determine whether there was some input to P that would return a link to playboy.com, and answer "yes" or "no." For any arbitrary program Q and input I, we could automatically translate it to a program Q´ that runs Q on I and outputs "playboy.com" whenever it halts. If Q halts on I, our program A will flag Q´ as unacceptable; if Q doesn't halt, our program A will approve Q´. We could thus use our analyzer program A to solve the halting problem. Because that's impossible, such an A (that works correctly for every program) can't exist.

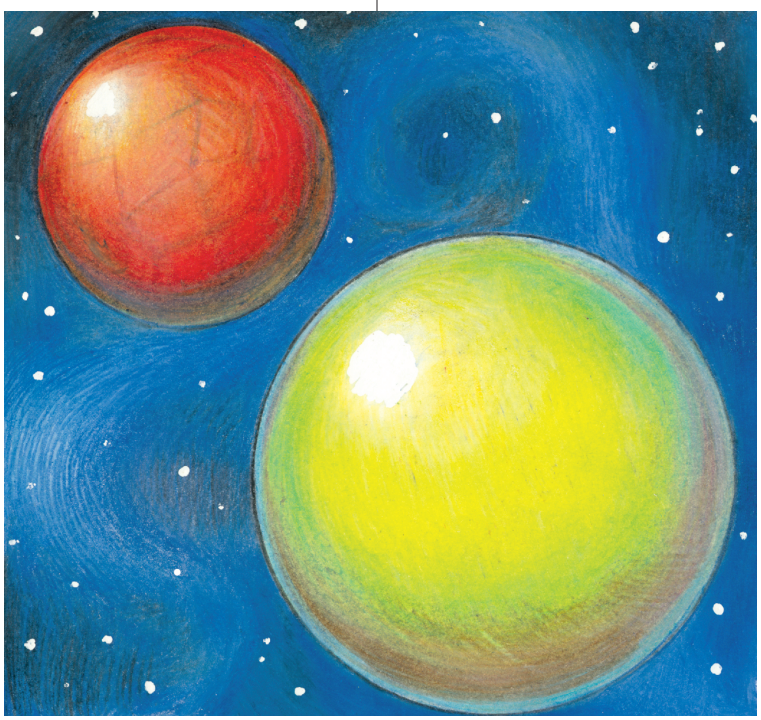
Chuckling, I cited Turing's 1937 paper[1] on the halting problem in my report for this agency. (I'm not sure if they renewed their contract with us; they probably wished I had given a practical answer instead of a correct one.)

Information theory

I had another, more recent, experience with computer science undermining a security project. When trying to attack an Internet system, it helps an adversary to know the details of the system's OS and underlying architecture because the adversary can then pick appropriate attack strategies. A student suggested defending systems by keeping adversaries from obtaining this knowledge, and (as an engineer) proposed measuring these techniques' effectiveness via *entropy*.

Entropy, in Shannon's information theory, measures how much information a message carries. (Some scientists characterize information content as how surprising a message is.) Although information theory is a beautiful way to reason about things such as transmission over noisy channels, it neglects to take into account the difficulty of actually extracting information. Consequently, in many practical scenarios, trying to minimize what the adversary might learn by minimizing entropy gives exactly the wrong answer because a substantial gap might exist between how much information a message contains and what the receiver can extract. (University lecturers certainly appreciate this phenomenon.)

For example, think about encryption with standard symmetric block ciphers such as the Triple-mode Data Encryption Standard (TDES) or the Advanced Encryption Standard (AES). An eavesdropping adversary can see the ciphertext, but knows neither the plaintext nor the key; to get this information, the adversary might try to guess the key and then use this guess to decrypt the ciphertext to a potential plaintext. If the plaintext



might be any possible bitstring, the adversary can never know if he or she guessed the right key. However, if the plaintext space is sufficiently sparse (for example, English text in ASCII characters instead of arbitrary binary) and if the ciphertext decrypts to valid plaintext, the adversary can be more confident that he or she guessed correctly. Shannon formalized this relation: the longer the ciphertext gets, the more confident the adversary can be that a guess is correct, if the ciphertext decrypts to sensible plaintext. Once the ciphertext is sufficiently long, it uniquely determines the plaintext. For encrypt–decrypt–encrypt TDES and English ASCII plaintext, as few as 18 characters are enough.

Suppose then that we want to keep the adversary from learning some data, such as machine configuration, that we express as 18 or more characters of English text, and we wanted to choose between two strategies. Should we delete every other character and send the remainder to the adversary? Or should we encrypt the entire string with TDES and a secret random key, and send the ciphertext to the adversary? If we use entropy to decide, the former is a better strategy. If we use common sense, the latter is better. Yes, we might have given the adversary sufficient information to compute the plaintext, but (by current intractability assumptions) the computation isn't feasible. Computer science and computer engineering give apparently contradictory answers—Turing is from Mars, Shannon is from Venus.

On the other hand, Shannon's information theory can also enable security projects. For example, consider trying to build a system that decrypts a ciphertext, encrypted under an unknown key, by systematically trying every key. The typical computer science student could build a decryption module and determine how to coordinate the activity of a vast collection of modules in parallel. Skilled in complexity theory, a computer science student might also be able to estimate the expected time and space complexity necessary for this search. However, a computer science education where Turing eclipsed Shannon wouldn't help the student tell, automatically, whether a guessed key is right.

A computer science education where Turing eclipsed Shannon wouldn't help the student tell, automatically, whether a guessed key is right.

For this problem, information theory gives the right answer. The point where the information in the ciphertext dominates the adversary's uncertainty about the key (for example, the approximately 18 characters in the previous case) is called the *unicity distance*. In the late 1990s, the Electronic Frontier Foundation (EFF) made a technical (and political) statement by building Deep Crack, a machine that used a massive number of such decryption modules in parallel to break DES by brute force.[2] It takes information theory to appreciate the technical slickness of the EFF design: essentially, each decryption module uses unicity distance to limit how much ciphertext it has to deal with, and a bit-vector to test for plaintext such as ASCII.

A dangerous fondness for clean abstractions

Computer science educations can also leave students ill-equipped in other ways. My colleague Michael Caloyannides (who has also written for *S&P*) has an entire list. One that stayed with me is the tendency of nonengineers to believe physical specifications (that is, I suppose, if they read them at all). For example, students tend to believe that if the spec for the wireless card says that the signal goes 50 meters, it can't be detected at 51 meters, and it can be detected perfectly at 49 meters.

This becomes a problem when such students use these assumptions to guide subsequent operational and policy decisions. The system they're securing isn't an abstraction following clean rules, but a mess of physical devices interconnected in the real world.

Overemphasis on clean abstractions hampers computer science students' security education in other ways. A continuing debate exists in computer science departments about what programming languages to use in the curriculum. Usually, this debate centers around which high-level language provides the cleanest abstraction to let new students learn basic good design principles and algorithms. Maybe I'm just being a middle-aged fogey here (having come of age in an era when the only way you could get a computer as a teenager was to design and build one yourself). However, by the time these students arrive in my operating systems class, I have to undo the clean abstractions they learned in their introductory courses, and instead teach them C and explain that their high-level languages reduce to compilers, OSs, syscalls, machine code, and transistors.

Students need to understand how these abstract functions, methods, variables, and so on—which have always magically appeared for them—are actual bytes living in actual memory. The abstraction of

high-level object-oriented languages might be necessary for learning good design principles, but stripping the abstraction away is necessary for understanding many important issues in performance, efficiency, implementation, and security. Students need both levels of understanding to be effective computer professionals.

For example, buffer overflow is still a major source of security problems in the information infrastructure. The "return-to-libc" variant (which caused a compromise at Dartmouth recently) is particularly nasty; the attacker doesn't even need to inject code, so marking the stack as nonexecutable doesn't help. Students suffer from these attacks today; when they graduate into the real world as computer professionals, their employers and customers will face these attacks. Without understanding how it all comes down to bytes in memory, students won't be equipped to handle these challenges.

Light bulbs needed

Moving from stack-smashing to abstraction-smashing, a few years ago, Princeton's Sudhakar Govindavajhala and Andrew Appel developed a wonderful example of the interplay between perspectives of understanding and security issues.[3] At an Internet level, a fundamental security challenge in the infrastructure is that many parties want to run code on your computer. How do you permit this functionality without also letting malicious or merely clumsy programmers trash your machine? One way to make it easier for programmers to do the right thing—and to assure the end user that these programs won't do the wrong thing—is to work in a programming language designed to make it (hopefully) impossible to write code that damages the rest of the system. This idea was behind Java Virtual Machines, but the Princeton researchers defeated this language-based security mechanism by filling memory with devious data structures that broke the protection if any one bit in a large fraction of memory spontaneously flipped. They then induced such spontaneous bit flips using a light bulb—the heat gently induced memory errors. For other examples of the real world breaking clean abstractions we need only consider the continuing effectiveness of side-channel attacks on cryptographic systems, such as a smart card on a lab table—or a Secure Sockets Layer server on the other end of the Internet.

Curmudgeons might observe that we could defeat the light bulb attacks on Java using error-correcting memory. Still, too few students see the whole picture—that is, the problem that type-safety can solve, the ways it can solve that problem, and the fact that light bulbs might be a risk. Turing is from Mars, Shannon is from Venus, and never the twain shall meet. We need to change that. ▢

References

1. A. Turing, "On Computable Numbers, With an Application to the Entscheidungsproblem," *Proc. London Mathematical Society*, ser. 2, vol. 42, 1937.
2. Electronic Frontier Foundation, *Cracking DES*, O'Reilly, 1998.
3. S. Govindavajhala and A. Appel, "Using Memory Errors to Attack a Virtual Machine," *Proc. IEEE Symp. Security and Privacy*, IEEE CS Press, 2003, p. 154.

S.W. Smith *is an assistant professor of computer science at Dartmouth College. Previously, he was a research staff member at IBM Watson, working on secure coprocessor design and validation, and a staff member at Los Alamos National Laboratory, doing security reviews and designs for public-sector clients. He received a BA in mathematics from Princeton University and an MSc and a PhD in computer science from Carnegie Mellon University. Contact him at sws@cs.dartmouth.edu; www.cs.dartmouth.edu/~sws/.*