Scott A. Craver

Assistant professor,
Electrical and Computer Engineering
Binghamton University
Binghamton, NY 13902-6000
phone: 607.777.7238
scraver@binghamton.edu

Teaching Statement

My teaching experience has been within the framework of information security.

Classes Taught: Since arriving at Binghamton I have designed and taught a slate of new courses pertaining to information security. First and foremost is a graduate course and a senior seminar on **cryptography**, providing a mathematical background for the remainder of the courses. Supplementing this is a senior seminar on **security engineering**, focusing on subjects not touched by a mathematical treatment of cryptography: policy and law, the design cycle and design principles, selected security problems in real life, including solutions and common attacks. We hope to turn our portfolio of courses into an explicit security curriculum; as of this writing our department already has a security category on our Ph.D. qualifying examination.

On the graduate level I also teach a course in **information theory**. This course is designed to be taught through our Enginet program, which makes the course available to distance-learning students as well as the local classroom.

Teaching Philosophy: I am largely non-interventionist, except when it comes to teaching mathematics. In pedagogical terms, *interventionism* implies that a teacher should reach out to students who are missing class, missing homework, or otherwise falling behind. Interventionism is the belief that instructors must identify and correct problematic behavior in underachieving students. The non-interventionist holds that a student is expected to work without prompting or cajoling, and indeed should be graded by his or her ability to do so. Employers want to know if students have the independence, maturity and work ethic to complete a task even if it is boring or difficult, and a student's grades should reflect that information.

What position one takes depends on the circumstances of courses taught. I teach senior-level and graduate courses; if I taught first or second year courses, or elementary prerequisite courses, I could have a different outlook. The subject matter also matters: in engineering or computer science, it makes sense to leave students to find their own path. In mathematics, even mathematically inclined students enter college with over a decade of ingrained math anxiety. Teaching that subject is as much psychology as anything else, and requires the instructor to reassure the students, and monitor them more carefully for problems. For that reason, when teaching the mathematical side of cryptography I am more of an interventionist in my teaching practices; in other courses (and in graduate courses) I insist that students bear the responsibility of work without prompting.

Teaching and Research: I cannot separate teaching from research. I fear for my productivity should I ever have a semester in which I do not teach. Each pass of coursework refines the ideas I use in research, and forces me to revisit the central principles behind them.

The security engineering course has been particularly valuable, and teaching it has provided deep insights. In particular, it has been very fruitful to analyze our previous attacks in search of general design principles, so that our "design principles" section can be more than Kerckhoff's Criterion and the Principle of Least Privilege. One valuable discovery is the phenomenon of abstraction leakage as the basis for almost all attacks and vulnerabilities. An abstraction leak occurs when a programming environment unexpectedly betrays details of its underlying implementation; a common example is variation of access times in a supposedly flat address space, caused by cache memory and paging.[1] Buffer overflows, format string specifier bugs, SQL injection attacks and cross-site scripting (XSS) attacks are all examples of abstraction leaks: in all cases, complex data is implemented as a flat string, and shortcomings in that implementation allow most attacks on computers today. If I were focused entirely on research I may never have noticed this bigger picture.

Contests and Competitions: I have recently come to appreciate the value of concrete challenges to students, in the form of competitions and contests. These are not meant to pit students against one another, but to challenge them collectively to achieve a concrete task. This was brought home to me after NAE Frontiers of Engineering symposium, where a researcher from NIST showed how an annual contest resulted in a substantial improvement in the state of the art of face recognition technology. The extracurricular nature of competitions can also motivate students who may otherwise

I am the creator and organizer of the annual Underhanded C Contest, a security contest to write simple but deceptive C programs. The contest is hosted at underhanded.xcott.com: every year participants are asked to write a short program that performs a simple data processing task, but conceals malicious behavior that is not obvious to someone looking directly at the code. The outcome is always educational, and illustrates the need for explicit code review.

As a professor in the Watson School of Engineering and Applied Science, I took it upon myself to start a Watson School Team Trivia Challenge, essentially a "pub quiz" for students with all questions coming from engineering, science and mathematics. The questions are not genuine trivia, but often serve a pedagogical purpose. I will typically ask questions that require estimation, or application of physics or chemistry. I also take the opportunity to ask test questions to ensure that students possess prerequisite knowledge. Despite the blatant pedagogical value of these contests, our turnout always exceeds expectations, if not the fire code for our meeting room.

---

[1] This idea was first described by Joel Spolsky as the "Law of Leaky Abstractions" in his book *Joel on Software*. Spolsky uses the law as a programming principle that affects software performance, exhorting young coders to remain aware of low-level phenomena when working within an API.