Teaching Computer Security

Kyle V. Moses and W. Michael Petullo
Department of Electrical Engineering and Computer Science
United States Military Academy
West Point, New York 10996
kyle.moses@usma.edu, mike@flyn.org

***Abstract*—Computer security is a tremendously challenging area of engineering. Our society finds itself increasingly reliant on computer systems, even while these systems regularly succumb to malicious attacks. Thus we must better prepare future engineers and scientists for the task of designing the new systems which will remain robust despite the threat environment found on the Internet. This paper suggests a new way to look at computer security education. It first presents a number of common shortcomings in computer security education. Next, it applies Bloom's taxonomy of educational objectives to the domain of computer security. Finally, it describes the educational experiences which will maximally benefit computer security engineers and scientists at the undergraduate level.**

I. Introduction

The routine art in the field of computer software construction does not yet produce software which preserves confidentiality, integrity, and availability when exposed to the Internet. This is evident as researchers continually find exploitable flaws in software which result in security vulnerabilities. The well-publicized Heartbleed [22] and Shellshock [23] vulnerabilities provide two examples, but the US National Vulnerability Database categorizes thousands of other new security vulnerabilities each year. It is not just uninformed end users who struggle with security. The best (and extraordinarily well-funded) intelligence agencies in the world have acknowledged that they consider portions of their networks compromised [34].

Security is a very difficult engineering problem. Evidence shows that the *design flaws* of systems confound the problem more than *implementation defects*. For example, it has been known for decades that trying to add security to an existing software artifact without a fresh design is counterproductive [13, 17]. Examples of design flaws include protections so complex that they make configuration and verification exceedingly difficult and protections that are omitted entirely. Yet *how to design* seems less understood than the implementation techniques described by the literature, and design also appears more dependent on Bloom's higher-order educational objectives.

The reason why design dominates implementation becomes evident if we consider an implementation flaw—and perhaps a security vulnerability that might arise from it—as a programmer misunderstanding. Examples of misunderstandings include an ill assumption about the semantics of a programming-language statement, an Application Programming Interface (API), or the size of an array defined using an unsafe programming language [2]. Programmer misunderstandings such as these arise due to the complex nature of software, but this complexity comes in two forms: essential and accidental [9]. The latter results from design decisions, and thus design decisions aid or hinder understanding.

For an example of a wise design decision, consider the choice to move array bounds checking from individual programs into a programming language's type system. This decision reduces accidental complexity, as it transforms the task of verifying the use of buffers in a number of applications written in an unsafe language to the task of verifying a type-safe programming language's implementation. This is significant because generally detecting buffer overflows using static analysis is an undecidable problem [20]. Better management of security problems involves envisioning better designs, and this involves the analysis and evaluation of existing and candidate designs against security requirements.

Too often, the results of mistakes in either design or implementation come to degrade the robustness of computer systems. The security of poorly-designed systems ends up overly dependent on end users who as a result of defective software must navigate through harsh security policy externalities and surprising software behavior. It is worth mentioning that defects are defects. While an intelligent adversary might more quickly cause a defect to manifest itself as a bug and subsequently happen to use the bug to his advantage, the defects that arise due to our insufficient understanding of the software we craft would remain and eventually cause malfunction even if the Internet were benign.

This paper is concerned about education. Education is the prelude to the efforts of the programmer described above. During the course of his education, a programmer must internalize the implementation techniques he encounters so that he is able to apply them to the requirements that arise in the future. Furthermore, he must understand current system designs sufficiently. The programmer will then be able to analyze and evaluate existing designs as he creates the future designs which will better manage the security problem.

Here we first describe related work (§II) before discussing

Public domain.
Permanent ID of this document: aa9579cdbbc08cb57418fedd5ded35b6.
Date: November 5, 2014.

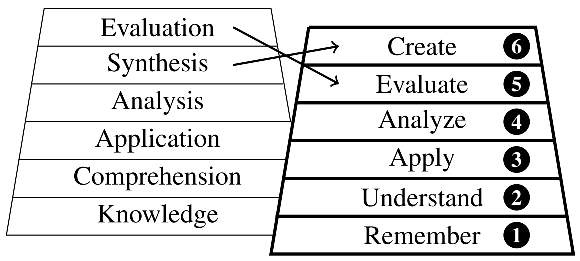

Fig. 1. Bloom's educational objectives; the original objectives appear on the left, and Anderson et al.'s revision appears on the right; we use Anderson's names here

some of the shortcomings present in computer science education as it applies to security (§III). We next apply Bloom's taxonomy of educational objectives [7, 3] to computer security education (§IV) in a different way than previous work, and we use our taxonomy to analyze student performance at a major computer security competition (§V). Finally, we extrapolate some key experiences necessary for a rigorous study of computer security (§VI). Here we argue that many components of security are necessary for general correctness, and thus these components ought to permeate an entire computer science curriculum.

II. Related work

Bloom et al. first published a taxonomy of educational objectives in 1956 which divided the functions of thought into six categories [7]. We depict Bloom's educational objectives in the left side of Figure 1. Anderson et al. revised Bloom's work in 2001 [3], and we depict this revision in the right side of Figure 1. This paper makes use of Anderson's revision.

Other work applies Bloom's objectives to both computer-science and computer-security education. Johnson et al. studied how computer-science program assessors and instructors characterized various evaluation events encountered by first-year computer-science students [18]. The authors found that assessors categorized evaluation events mainly at meeting Bloom's objectives remember–apply. While the instructors categorized more events as demonstrations of analysis, they too agreed that the most commonly met objective was application. Following this study, the authors propose a new objective—*higher application*—which represents application, but only if informed by analysis, evaluation, and creation.

A project at Pacific Northwest National Laboratory developed a series of computer-security awareness training materials, and they posited that their training exercises Bloom's first three objectives: remember–apply [25].

Van Niekerk et al. applied Bloom's objectives to the security training and education of end users. [21]. In this work the authors posit that organizational end users must understand *why* information security policies are necessary if they are to restrain themselves to act within those policies. Further, the author presented a number of sample questions and exercises that would evaluate which of Bloom's objective are met by a student's understanding of information security.

Buchanan et al. characterized their interactive cyber-security training products vis-à-vis Bloom's objectives [10]. Their training modules instruct students on the use of tools such as `nmap`, the de facto correspondence between transport-layer ports and services, and how to manage the robust composition and operation of existing network products.

III. Shortcomings in computer security education

What is limiting in the studies described above is that each focuses on end users. For example, van Niekerk states that if only users understood *why* passwords were important, then they would choose and protect them with more care. However, this is an example of trying to overcome the inadequacies of system design by training end users. Such a reaction ignores that passwords alone are a weak form of authentication which many researchers claim ought to have a very limited place in modern systems [8].

We consider here the security education of computer scientists, as opposed to end users. It is well known that even trained users routinely ignore security-related prompts which often get in the way of their work [16]. Instead, we address the larger role that the design and implementation of computer systems has on their eventual use, and how to better educate computer scientists so that they can to achieve these designs and implementations.

To illustrate why computer science education is currently insufficient, let us first consider the end user who is the subject of PNNL, van Neikerk, and Buchanan's studies. Often, such a user receives the blame for security breaches because he did not choose a strong password, because he executed a malicious program attached to an email, because he transmitted information over an unauthenticated and unencrypted network connection, or because he mistakenly trusted one website which poses as another after clicking on a link in an email. Indeed many training materials warn users about each of these actions; for example the US Department of Defense (DoD) Cyber Awareness Challenge Training teaches students that passwords ought to be composed of 15 characters—one lower-case, one upper-case, one number, and one punctuation mark—but that they should not be written down (does anyone think this is a viable system?). The same training warns users against placing unknown CD media in their computer (why does potentially malicious code have access to information that could cause harm?), and it urges them not to trust links in emails (why is the source of information being presented to the user not always clear?). In the economy of individual end users—who consider the benefit of following security policies as disproportional to their cost—ignoring these requirements is rational.

Security researcher Dan Bernstein stated:

> My views of security have become increasingly ruthless over the years. I see a huge amount of money and effort being invested in security, and I have become convinced that most of this money and effort is being wasted. Most "security" efforts are designed to stop yesterday's attacks but fail

> completely to stop tomorrow's attacks and are of no use in building invulnerable software. These efforts are a distraction from work that *does* have long-term value [4].

Managers, programmers, and administrators who blame end users or systems whose security depends on end user education will always fail. Instead, computer scientists ought to design their systems to use authentication techniques which are stronger and more convenient than passwords, to appropriately constrains untrusted programs, to forbid unauthenticated or unencrypted network connections, to ensure that the source of information presented to a user is always evident, and so on. Furthermore, architects should design systems that provide these properties across all programs, no matter how the individual programs themselves are written.

Why then, does the education of our programmers and system administrators fail to produce in sufficient numbers professionals who are able to construct systems that survive contact with the Internet? Why are systems so frail that they require end users to often work in counter-productive ways and deal with unnatural interfaces in order to maintain an organization's security requirements? We posit that an insufficient number of programmers achieve the higher-orders of Bloom's educational objectives vis-à-vis security (and general system design and implementation), and thus we lack enough programmers and system administrators who can sufficiently reason about the systems they design and produce. This is compounded by the fact that modern Operating Systems (OSs) are unnecessarily complex and leave too much security functionality to each application;[1] this too is a design flaw which results from insufficient analysis and evaluation. Together, these factors impede our ability to construct systems which are both innovative and robust at the pace expected by the information age.

What is it then that practitioners ought to know? Figure 2 describes these components. Robust programming requires the protection of confidentiality, integrity, and availability. Such protections result from the composition of a number of security services, such as authentication and authorization. This composition in turn is guided by a number of principles, including domain separation, least privilege, verifiable construction, tamper-proof access controls, natural interfaces, and complete mediation. These principles can be traced back to the principles of Multics [12] and early work by Rushby [30].

IV. Bloom's taxonomy

From an educational standpoint, security failures are due to practitioners attempting to manage the security problem at the less-sophisticated levels of Bloom's hierarchy. A number of commonly-applied techniques demonstrate this flaw: firewalls ignore that transport-layer ports have at best a de facto relationship with services (and that furthermore protocols such as HTTP can represent any number of services); antivirus scans

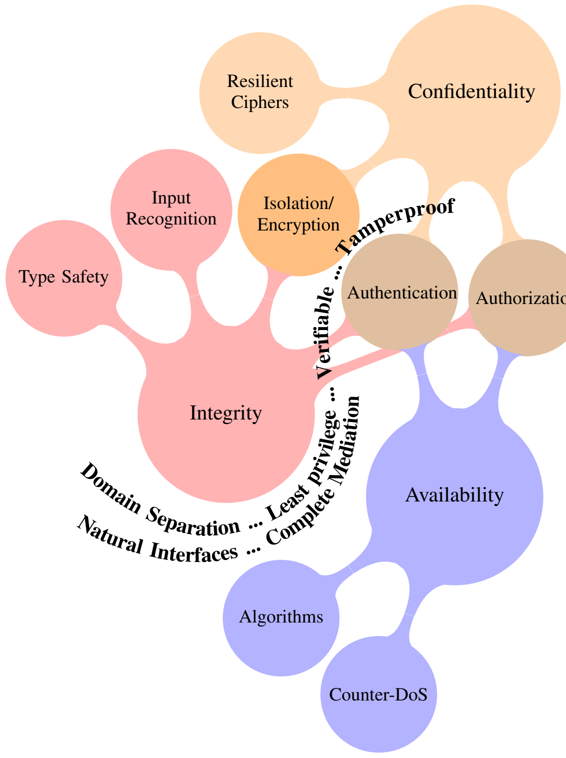

Fig. 2. Requirements of robust design; robustness requires the protection of confidentiality, integrity, and availability; a number of services—such as authentication and authorization—support these goals, and principles such as domain separation and least privilege guide good designs

have difficulty catching anything other than known threats; and keeping software up to date does not remove undiscovered flaws, which leaves systems vulnerable to unknown attacks. Heartbleed and Shellshock also represent naive designs. The harm from the former was compounded because applications were not broken into mutually untrusting, isolated components (i.e., only a small process ought to have access to secret encryption keys). The latter caused harm due to the unwise decision to build servers from sophisticated software[2] designed for interactive-command-line use.

Table I describes one progression through Bloom's objectives as a student learns about the security services listed in Figure 2. It is worth noting that the table is not exhaustive; instead it provides samples which represent the level of knowledge to which educational experiences must bring students should they be prepared to craft robust software.

V. Case study: analyze a security exercise

The Cyber Security Awareness Week (CSAW) Conference, hosted annually by the New York University Polytechnic School of Engineering, includes a Capture The Flag (CTF) competition specifically targeted at the entry/undergraduate level. The CSAW CTF is a Jeopardy-style competition in which teams race to complete security challenges in multiple categories which vary in difficulty and point value. CSAW's

[1]Work to better understand this includes the Ethos project [26, 28, 27]

[2]The bash shell uses 99,000 lines of C code.

TABLE I
SAMPLE LEARNING OBJECTIVES FOR THE REQUIREMENTS OF SECURE DESIGN

	Authentication (Confidentiality, integrity, and availability)
Remember	Describe authentication as the process of identifying principles so they can be subject to subsequent access controls
Understand	Give examples of knowledge, possession, and inherence factors used of authentication
Apply	Implement a computer program that makes use of authentication
Analyze	Contrast the advantages of different authentication factors when used for local and network authentication
Evaluate	Explain why so many computer programs fail to properly authenticate the principle upon whose behalf they run
Create	Design a trusted computing base that ensures all untrusted programs run with the privileges appropriate for their principle
	Authorization (Confidentiality, integrity, and availability)
Remember	Describe authorization as the process of restricting computer programs so that they perform only permitted operations on permitted objects
Understand	Distinguish the difference between discretionary and mandatory access controls
Apply	Implement a program which constrains users through the use of access controls
Analyze	Contrast different means of access controls, for example, flow controls, type enforcement, and capabilities
Evaluate	Argue the merit of system-wide access controls when compared to application-specific access controls
Create	Design a series of operations, objects, and access controls which achieve complete mediation
	Isolation/Encryption (Confidentiality and integrity)
Remember	Define isolation as a means of constraining access to a resource
Understand	Understand that information has an appropriate scope and that isolation can enforce its scope
Apply	Use existing isolation mechanisms such as processes and encryption to produce a program which constrains access to a critical resource
Analyze	Categorize different techniques for achieving isolation
Evaluate	Defend the value of mandatory encryption and isolation
Create	Create a series of system interfaces which dictate the construction of applications as isolated components
	Algorithms (Availability)
Remember	An algorithm has a theoretical performance which can be described in terms of asymptotic notation
Understand	Explain the performance difference between different asymptotic bounds
Apply	Choose an appropriately performing algorithm for some performance metric
Analyze	Compare the theoretical throughput of a network link/protocol to the resource consumption required by the algorithms and data structures employed by a service
Evaluate	Appraise the merits of different algorithm and data structure choices vis-à-vis theoretical throughput
Create	Produce a service implementation which is reasonably robust against Denial of Service (DoS) attacks
	Input Recognition (Integrity)
Remember	Recognize the need for input validation
Understand	Describe how a lack of input validation gives rise to security vulnerabilities
Apply	Discover an input validation failure in a computer program
Analyze	Relate the input validation problem to parsing
Evaluate	Explain the need for formal-grammar definitions of program inputs and outputs, that is that the context-free equivalence problem is undecidable
Create	Define a formal I/O grammar and use a parser generator to create the input validation routines for a computer program
	Type Safety (Integrity)
Remember	Describe type safety as constraining the operations permitted to act on a region of memory
Understand	Describe why attacks such as stack-smashing attacks and defects such as integer overflow demonstrate a lack of type safety
Apply	Show that using a verified type-safe language is easier than avoiding buffer-overflow bugs as you implement a set of programs
Analyze	Identify the relationship between type safety and input recognition
Evaluate	Appraise the merit of various techniques to achieve type safety and tie type safety to input recognition
Create	Create a series of system interfaces which bridge the semantic gap between individual-program type systems and system-level code
	Counter-DoS (Availability)
Remember	Describe DoS countermeasures such as puzzles
Understand	Explain how amplification attacks arise from protocol misdesign
Apply	Employ puzzles to raise the cost of performing a denial of service attack on a network protocol
Analyze	Catagorize a number of protocols with respect to amplification-attack vulnerability
Evaluate	Design a new protocol whose availability can be disrupted only by an attacker who performs an excessive amount of work
Create	Implement a DoS-resilient system, and perform empirical experiments to test its resiliency
	Resilient Ciphers (Confidentiality)
Remember	Recognize that implementation artifacts—such as the varying effect of caching on performance—can leak information about cryptographic keys
Understand	Describe why table-based cipher implementations often succumb to side-channel and other attacks
Apply	Show a side-channel attack on an existing cipher
Analyze	Breakdown a number of cryptographic protocols to thier primitive components
Evaluate	Compare the merits of a number of ciphers
Create	Implement a system which makes proper use of ciphers and other cryptographic primitives

categories this year included: exploitation, forensics, web, reconnaissance, networking, cryptography, reverse engineering, and trivia. Unlike the namesake game show, the competitors submit solutions (known as flags) confidentially. This scoring model allows all teams to solve a challenge independently and still earn points, and this maximizes participation and the educational value of the competition.

According to the CSAW website [24], the qualifying round for 2014's competition (which occurred in September) included 998 teams from around the world. 471 of these teams included undergraduate members with the remainder composed of graduate students, industry professionals, high school students, and others with no declared affiliation. During the qualifying round there were no restrictions on team size or composition and while no count of team size was released, the CTF site was accessed by 18,713 unique IPs, which yields a rough estimate of total participation. The precise educational background of individual participants is unknown, but this information is not vital to our analysis since we can still draw meaningful conclusions about the overall state of computer security education from the aggregates. This contest format consequently serves as an excellent dataset to evaluate using our proposed security education model.

The results of this year's qualification round are publicly available through the statistics page of the CSAW CTF website [24]. Additionally, the website includes solution write-ups for each challenge, produced by participants who completed them successfully. In our analysis, we used the challenge descriptions and their associated write-ups to qualitatively map each challenge to our Bloom's Taxonomy-based learning objectives from Table I. The Reconnaissance and Trivia categories were excluded from our analysis since they typically had weak or no correlation to computer science learning objectives.

As an example, Forensics Challenge 28 "Fluffy No More", required each team to analyze the logs and a database backup of a web forum to figure out how the site itself was compromised and how the attacker then exploited the systems of visiting users. Analyzing the logs revealed that the attacker conducted a brute force fuzzing attack against the web server to search for vulnerable plugins. From this point, the attack used the vulnerability to compromise the server by uploading a JavaScript file which used a PDF exploit as an attack vector against site users. The JavaScript code and PDF embed were both obfuscated and the PDF itself included an obfuscated version of the challenge flag.

This challenge scenario required participants to compose the solution through multiple level two and three learning objectives. The primary exploit used was an input validation failure which exposed system credentials to the attacker that were not properly isolated. A thorough understanding of Linux authentication and authorization mechanisms was necessary to understand how the attacker gained control of the full system after the initial compromise. This same interplay of learning objectives was at work in the PDF exploit that the attacker turned against the site users. Finally, a significant red herring left by the attacker was the encrypted password in the shadow file. A participant might have assumed the password to be a key value that needed to be cracked, but a quick analysis of system settings and knowledge of the strength of SHA-512 would prevent a participant from wasting time on this account credential.

After completing this qualitative analysis, we then compared the taxonomy levels associated with each challenge to the completion rates posted on the CSAW CTF website. The rates were derived from aggregate results including the performance of graduate and industry professional teams. This is not considered a problem since it only serves to make our observations at worse err on the conservative side. Our results are depicted in Table II. The numeric values in the table directly correspond to the Bloom's Taxonomy levels from Figure 1. The horizontal dividing lines represent the normal distribution of challenge completion rates and mark the median and standard deviation boundaries. The table includes the official CSAW Challenge IDs to allow others to perform further analysis.

The vast majority of challenges met objectives corresponding to levels 1–3 (remember–apply) in one or more categories. Additionally, the challenges with the lowest completion rates (-1 to -2σ) typically involved multiple learning objects at levels 3–4 (apply–analyze). The table also draws attention to the limited presence of challenges directly involving the important categories of Authentication, Authorization, and Counter-DoS as well as the complete absence of challenges mapping to level 5–6 (evaluate–create) in any category. These observations should not be taken as a criticism of CSAW, especially since this event is targeted at entry level students and since our analysis to this point has only looked at one instance of the competition. The results simply draw attention to the fact that a CTF on its own would not serve well as a capstone experience in a security education program for computer scientists. The results also show that the community at large—including graduate students and industry professionals—often still struggles with level 3 and 4 objectives. We posit that the solution to this struggle will come through adequately addressing level 5 and 6 objects in the educational system.

VI. Necessary educational experiences

Whether security should permeate an entire computer science curriculum is somewhat controversial, with some opponents arguing for a new computer security major[3]. We argue that a distinct major would fail to prepare students to manage the security problem, *because the solutions we need will come from the application of computer science*. If our designs are insufficient, and it is not possible to fix the security of a system without a redesign, then a reliance on teaching practitioners how to implement, configure and manage current designs is irresponsible and doomed to fail.

We must educate those who will see further than us. Instead of forming a distinct major, portions of a computer security education ought to permeate through an entire computer science

[3]A number of universities provide a Cybersecurity or Information Assurance major, the US Naval Academy provides a Cyber Operations major, and the US Military Academy is likely to consider a similar major.

TABLE II
CORRELATION BETWEEN CHALLENGE COMPLETION RATES AND TAXONOMIC LEARNING OBJECTIVES

Challenge ID	Category	Completion Rate	Authentication	Authorization	Isolation	Algorithms	Input Recognition	Type Safety	Counter-DoS	Resilient Ciphers
22	Exploitation	87.1%	0	0	2	0	0	0	0	0
2	Forensics	74.0%	0	0	2	0	2	0	0	0
12	Networking	64.6%	0	0	2	0	2	0	0	0
1	Forensics	64.1%	0	0	1	0	2	0	0	0
	Forensics	6 .6%	0	0	1	0	3	1	0	0
	Reversing	5 .0%	0	0	0	0	2	0	0	0
7	Exploitation	44.4%	0	1	2	0	1	0	0	0
6	Reversing	4 .7%	0	0	2	0	0	0	0	0
8	Cryptography	22.0%	1	0	3	3	0	0	0	3
24	Exploitation	17.9%	0	0	2	0	3	3	0	0
21	Web	16.6%	0	0	2	0	3	0	0	0
28	Forensics	14.9%	3	3	3	0	3	0	0	2
27	Reversing	10.2%	0	0	0	0	3	2	0	0
0	Reversing	9.2%	0	0	2	3	3	0	0	0
10	Exploitation	7.2%	0	0	3	0	0	0	0	0
25	Exploitation	6.1%	0	0	0	0	3	4	0	0
9	Cryptography	5.8%	0	0	3	3	0	0	0	3
2	Exploitation	4.0%	0	0	3	0	3	3	0	0
2	Exploitation	4.0%	0	0	3	0	3	4	0	0
1	Cryptography	2.4%	0	0	3	3	0	0	3	4

TABLE III
ESSENTIAL CS-CORE AND SECURITY EDUCATIONAL EXPERIENCES

Experience	CS Core	Security
Domain Separation	✓	
Least privilege	✓	
Verifiability	✓	
Tamperproof design	✓	
Natural interfaces	✓	
Complete mediation	✓	
Machine architecture	✓	
Language theory	✓	
Compiler construction	✓	
Cryptology		✓
Algorithms	✓	
Operating system access controls	✓	
Operating system design		✓
Secure programming practices		✓

program, while others should be left to a specialized thread through the program which students might choose to pursue. Table III captures whether educational experiences belong in the computer science core or security thread.

***Guiding design principles*:** Table I does not directly address the guiding design principles we listed in Figure 2. Instead, these principles are what ought to permeate the educational experiences which *all* computer science students encounter. This is because these principles are equally relevant in the pursuit of correctness and security. Without the mastery of these principles, a practitioner will inevitably create programs that he does not understand, and these programs will neither be correct nor secure.

For example, computer programs ought to remain as small as possible (while preserving their intended function) in order to permit their verification (if a sound design could be found, then the verification of its implementation would guarantee robustness). Ken Thompson famously said, "One of my most productive days was throwing away 1,000 lines of code." Dijkstra put it another way: "software managers measure 'programmer productivity' in terms of 'lines of code produced,' whereas the notion of 'lines of code spent' is much more appropriate [14]." A student's natural inclination might be that large code counts constitute impressive programs, but they should be taught that elegance and simplicity are more virtuous. Such an understanding benefits both correctness and security.

But how can students arrive at designs which result in small implementations? Much of the requirements here are shared with general software engineering. UNIX provides a time-tested example of how to compose large programs from manageable units [29]. Even traditionally security-related topics such as domain separation have broad merit. For example, the Biba access-control model [6] is equally useful whether low-integrity data is the result of malicious attacks, honest mistakes, or timeliness requirements.

Least privilege is a deceptively complex guiding principle, but one that is necessary for both reliability and security. Bernstein describes why least privilege is often necessary but not universally sufficient for robustness [4, §5.2–5.3]. The problem arises because the least privilege necessary to perform some unit of work might already be enough to violate security. Thus a programmer needs to be able to reason about this and reinforce least privilege with other techniques when required (e.g., Bernstein's multi-source transformations require special care). Despite the effort required to achieve the background necessary to perform this analysis, it is a critical skill for the general computer-science population. It is well known that least privilege aids general system reliability; for example microkernels place an emphasis on constraining portions of an OS in user-space processes to avoid system-wide faults [33].

Natural interfaces do not frustrate users, but security is wrought with unnatural interfaces. Perhaps the call to isolate computer security studies in a distinct major arises from the false assumption that it has to be the case that security implies unnatural interfaces. In fact, it is our primitive understanding of computer security that gives rise to unnatural interfaces and other counterproductive burdens on end users. Better security design will result in more natural interfaces. Clearly, interface design is a principle which benefits both security and other subfields of computer science.

Perhaps the most elusive guiding principle is complete mediation. Complete mediation refers to a system's ability to restrict *all* system object accesses in accordance with some tamper-proof security policy. System designs must achieve complete mediation while permitting efficacious policies that can be understood by policy engineers. This is difficult, as success follows only the ideal intersection of each of the security services. A persistent question is: what level of abstraction should system objects assume to provide both generality and simple policy rules?

Few systems sufficiently approach complete mediation, and those that do serve a rather specific purpose. Only by meeting the highest levels of Bloom's educational objectives, will we produce practitioners who are better able to imagine the designs necessary to better approach complete mediation.

On its surface, complete mediation appears to be a security-specific goal, as it seems to imply OS access controls. However, its essence is shared with fundamental principles of reliability, namely enforcing abstractions. Consider language keywords such as Java's public and private. The purpose of these constructs is to promote well-defined interfaces and to restrict access that would otherwise violate the resulting abstractions.

***Critical subjects*:** Beyond mastering generally-applicable guiding principles, a computer security thread within a computer science program must further educate students on a number of critical subjects. Many of the topics we list here also benefit general computer science students.

It is important for computer scientists to understand the semantics of the interfaces upon which they construct their software. This includes understanding hardware, and thus a computer security thread must provide experiences in assembly-level programming. Events such as the CSAW CTF competition serve to provide experiences corresponding to the middle levels of Bloom's hierarchy. However, students must be able to characterize general solutions to the defects they exploit during the course of competing in a CTF event. Put another way, learning some processor's instruction set is necessary but not sufficient—after all, instruction sets eventually become obsolete. More importantly, students should draw general conclusions about how to best manage the limitations of hardware, for example by enforcing various forms of type safety at the programming-language level.

Language theoretic security posits that programmers must define the protocols which their programs make use of using formal grammars. They should then machine-derive recognizers—such as by using YACC, PADS, or hammer [15, 1]—if they are to make their programs trustworthy [31]. This is because the context-free equivalence problem—essentially the ability determine if two context-free grammars describe the same language—is undecidable.

A sufficient computer security education thus requires undergraduate-level language-theory and compiler courses. Practitioners must understand Chomsky's hierarchy of languages [11]. Without this knowledge, programmers will construct defective software. For example, programmers might process HTML using regular expressions or worse [32]. Furthermore, without a strong understanding of how to define formal grammars—along with how to derive lexical analyzers and parsers from a formal grammar—their programs will inevitably misinterpret the inputs they receive.

A study of cryptology—to include both cryptography and cryptanalysis—is necessary to satisfy the resilient ciphers security service, as well as the encryption portion of the isolation service. This is an example of a set of skills which is less relevant to general computer scientists. However, general programmers will benefit from the better security interfaces that this education will permit.

The purpose of studying cryptology here is not to design ciphers or even to choose the details of cipher use (we follow the authors of NaCl [5] who posit that a cryptographic library designed by cryptographers ought to choose the cipher suite programs will use). Rather, security programmers need to understand the purpose of the various cryptographic primitives (e.g., hashing, authenticated encryption, etc.). Able to make proper use of cryptographic primitives, security programmers should—in the spirit of NaCl—produce designs which make security decisions for programmers, in those places where the decisions do not unnecessarily impede generality. The result is a simpler system, a more clearly defined trusted component, and better-informed, more-consistent access controls.

A design's resilience against DoS attacks and thus the design's ability to protect availability is a function of the algorithms employed by the design. Particular DoS countermeasures—such as puzzles [19]—also require complexity analysis. Thus an algorithms course must be included in a computer security education.

Traditional security services such as isolation, authentication, and authorization should be taught in the context of a basic OS course for the general computer-science population. However, students with a security focus need more. It is very likely that many of the requirements of robust design will require modifications to OS kernels. This is because the OS kernel (or perhaps a hypervisor) is in a unique position vis-à-vis complete mediation—covert channels aside, all of a program's interaction outside of its address space must pass through the OS. For this reason, students of security should experience an advanced operating system course which goes beyond what generally is taught at the undergraduate level. Topics should include surveying existing OS system call interfaces and designing system call interfaces which will better aid programmers as they construct robust software. Along with an advanced OS course, a computer security curriculum should provide a course on secure programming practices.

VII. Conclusion

This paper focused on the education of computer scientists—the future designers and programmers who will inherit the security problems we face. It is clear that our current software base is failing to provide the confidentiality, integrity, and availability protections required. This security problem cannot be solved through composition and management alone. Further, it is evident that design will dominate implementation when considering solutions.

What should be clear—based on the necessity for topics such as algorithms, compilers, language theory, robust programming practices, and operating systems—is that the security problem is one that demands trained computer scientists. Creating a distinct degree path will inevitably miss providing students with key insights. Focusing on how to configure or fix implementation errors in present systems—which have been demonstrated to be insufficient by design—is irresponsible.

Future work ought to describe the educational experiences required in preparation for other career paths, such as system administration and project management. Of particular interest is how to reconcile the insufficiency of present system designs with the applied focus of the manners of computing education outside of computer science. Yet another area which requires thought is what, if anything, should be taught to the general population of students at the undergraduate level. We have not yet considered this to a high degree, but it seems that here the traditional tradesman-style approach of security awareness instruction and training is not a good fit.

We require computer scientists with the education necessary to produce a new generation of designs which are robust enough to survive contact with the Internet. Thus it is necessary to rethink computer science education to provide the experiences which support the higher-order Bloom's objectives needed for this task. While not all students will migrate towards focusing on robust design and implementations during their studies, all computer science students will benefit from studying the components of computer security which are shared with general correctness. This paper proposed a baseline of guiding design principles known to all computer scientists, along with an optional computer security thread which would result in a deeper understanding of a number of key security services.

References

[1] Hammer parser generator. https://github.com/UpstandingHackers/hammer [Accessed Feb 5, 2014].

[2] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine 7*, 49 (1996), File 14.

[3] Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Cruikshank, K. A., Mayer, R. E., Pintrich, P. R., Raths, J., and Wittrock, M. C. *A Taxonomy for Learning, Teaching, and Assessing: a Revision of Bloom's Taxonomy of Educational Objectives*. Longman, New York, NY, USA, 2001.

[4] Bernstein, D. J. Some thoughts on security after ten years of qmail 1.0. In *Proceedings of the 2007 ACM workshop on Computer security architecture* (New York, NY, USA, 2007), CSAW '07, ACM, pp. 1–10.

[5] Bernstein, D. J., Lange, T., and Schwabe, P. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America* (2012), vol. 7533 of *Lecture Notes in Computer Science*, Springer, pp. 159–176.

[6] Biba, K. Integrity considerations for secure computer systems. Tech. Rep. TR-3153, MITRE Corp, Bedford, MA, 1977.

[7] Bloom, B. S., Engelhart, M. B., Furst, E. J., Hill, W. H., and Krathwohl, D. R. *Taxonomy of Educational Objectives. The Classification of Educational Goals. Handbook 1: Cognitive Domain*. David McKay, New York, NY, USA, 1956.

[8] Bonneau, J., Herley, C., van Oorschot, P. C., and Stajano, F. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, May 2012), IEEE Computer Society Press, pp. 553–567.

[9] Brooks, F. P. No silver bullet. *IEEE Computer 20*, 4 (Apr. 1987), 10–19.

[10] Buchanan, L., Wolanczyk, F., and Zinghini, F. Blending Bloom's taxonomy and serious game design. In *Proceedings of the 2011 International Conference on Security and Management* (July 2011), SAM '11, WORLDCOMP.

[11] Chomsky, N. Three models for the description of language. *IRE Transactions on Information Theory 2* (1956), 113–124.

[12] Corbato, F. J., Saltzer, J. H., and Clingen, C. T. Multics—the first seven years. In *Spring Joint Computer Conference* (1972), AFIPS Press, pp. 571–583.

[13] Department of Defense. Trusted computer system evaluation criteria. Tech. Rep. DOD 5200.28–STD, U. S. Department of Defense, 1985.

[14] Dijkstra, E. W. Introducing a course on mathematical methodology. https://www.cs.utexas.edu/users/EWD/ewd09xx/EWD962.PDF [Accessed Oct 9, 2014], June 1986.

[15] Fisher, K., and Gruber, R. PADS: a domain-specific language for processing ad hoc data. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 295–304.

[16] Herley, C. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *Proceedings of the 2009 Workshop on New Security Paradigms Workshop (NSPW'09)* (New York, NY, USA, 2009), ACM, pp. 133–144.

[17] ISO/IEC Standard 15408. *Common Criteria for Information Technology Security Evaluation*, version 3.1 ed., July 2009. http://www.commoncriteriaportal.org/cc/.

[18] Johnson, C. G., and Fuller, U. Is Bloom's taxonomy appropriate for computer science? In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006* (New York, NY, USA, 2006), Baltic Sea '06, ACM, pp. 120–123.

[19] Juels, A., and Brainard, J. G. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the 6th Network and Distributed System Security Symposium* (Reston, VA, USA, Feb. 1999), The Internet Society.

[20] Larochelle, D., and Evans, D. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium* (Berkeley, CA, USA, Aug. 2001), USENIX Association.

[21] Niekerk, J. V., and von Solms, R. Bloom's taxonomy for information security education. In *Proceedings of the ISSA 2008 Innovative Minds Conference* (Pretoria, South Africa, July 2008), ISSA '08, ISSA.

[22] NIST National Vulnerability Database. CVE-2014-0160. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160, Dec. 2013. [Accessed Apr 15, 2014].

[23] NIST National Vulnerability Database. CVE-2014-6271. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271, Sept. 2014. [Accessed Oct 5, 2014].

[24] NYU Polytechnic ISIS Laboratory. CSAW'14 CTF. https://ctf.isis.poly.edu/ [Accessed Oct 9, 2014], Sept. 2014.

[25] Pacific Northwest National Laboratory. Cyber security awareness training. http://www.nwacc.org/programs/awards/excellence_award/pnnl_submissions_07/pnnl_cybersecurity_awareness_training.pdf [Accessed Oct 3, 2014], Dec. 2007.

[26] Petullo, W. M., and Solworth, J. A. Simple-to-use, secure-by-design networking in Ethos. In *Proceedings of the Sixth European Workshop on System Security* (New York, NY, USA, Apr. 2013), EUROSEC '13, ACM. https://www.ethos-os.org/papers/.

[27] Petullo, W. M., Solworth, J. A., Fei, W., and Gavlin, P. Ethos' deeply integrated distributed types. In *Proceedings of the 2014 IEEE Security and Privacy Workshops* (New York, NY, USA, May 2014), IEEE.

[28] Petullo, W. M., Zhang, X., Solworth, J. A., Bernstein, D. J., and Lange, T. MinimaLT: Minimal-latency networking through better security. In *Proceedings of the 20th ACM Conference on Computer and Communications Security* (New York, NY, USA, Nov. 2013), CCS '13, ACM.

[29] Ritchie, D. M., and Thompson, K. The UNIX time-sharing system. *Communications of the ACM (CACM) 17*, 7 (1974), 365–375.

[30] Rushby, J. Design and verification of secure systems. In *Symposium on Operating System Principles (SOSP)* (1981), pp. 12–21.

[31] Sassaman, L., Patterson, M. L., Bratus, S., and Locasto, M. E. Security applications of formal language theory. *IEEE Systems Journal 7*, 3 (2013), 489–500.

[32] Sassaman, L., Patterson, M. L., Bratus, S., and Shubina, A. The halting problems of network stack insecurity. *;login: the USENIX Association newsletter 36*, 6 (Dec. 2011), 22–32.

[33] Tanenbaum, A. S., and Woodhull, A. S. *Operating Systems: Design and Implementation 3/e*. Prentice Hall International, 2006.

[34] Zorz, Z. NSA considers its networks compromised. http://www.net-security.org/secworld.php?id=10333 [Accessed Sep 23, 2014], Dec. 2010.