COMPUTER SCIENCE EDUCATION:

SECURE SOFTWARE

By

JAMES FRANCIS CAIN III

Bachelor of Science in Electrical Engineering
University of Missouri - Rolla
Rolla, Missouri
1996

Master of Science in Computer Science
University of Missouri - Rolla
Rolla, Missouri
1999

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
December, 2010

COMPUTER SCIENCE EDUCATION:

SECURE SOFTWARE

Dissertation Approved:

Dr. Blayne E. Mayfield

Dissertation Adviser

Dr. John Chandler

Dr. George E. Hedrick

Dr. Marilyn Kletke

Outside Committee Member

Dr. Mark E. Payton

Dean of the Graduate College

ACKNOWLEDGMENTS

The author would like to thank several groups and many individuals for their support and encouragement, far too many for the author to be able to name all who rightly deserve to be mentioned. The author’s friends, family, colleagues, employer, supervisors, and professors all deserve immense thanks; the author would never have completed this doctoral program without their incredible assistance. A number of the author’s students, including all those who consented to participate in the human subject research, also deserve thanks for supporting and encouraging even as the author’s studies influenced and impacted their own studies.

A few individuals must be named for particular assistance, though. Dr. Blayne Mayfield, Dr. George Hedrick, Dr. John Chandler, and Dr. Marilyn Kletke all provided immeasurable assistance with laying the foundations for this dissertation and then focusing the author’s research as members of the author’s committee. Dr. Tim DeClue, Prof. Jeff Kimball, and Dr. Baochuan Lu provided tremendous support in reducing the author's workload during the research and the writing of the dissertation. The late Dr. Bill Warde served as a consultant for statistical methods to analyze the survey data. Dr. Troy Bethards served as a consultant for related business theory. Milton Austin, one of the author’s best friends and fellow OSU CS graduate student, worked tirelessly as a sounding board for eight years of the author’s brainstorming. Carissa Barker, the grammatical editor for the first draft of the dissertation, deserves recognition as well; may she someday recover from reading such poor grammar and may she someday forgive the author for all the mistakes he made in the subsequent revisions.

Finally, the author frequently has been heard to say over the past several years “Lord willing, I will yet finish this doctorate… before it finishes me!” Well, it is just about finished, so let the last but most important credit go to where it is most significantly due: thank you Lord!

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER I

INTRODUCTION

As the Internet has increased greatly, the means both to access and to distribute both code and data, computer security problems have become significantly more rampant and problematic. The Internet is not the cause of all of the individual security issues of the world's computer systems; it merely magnifies the scope of the problem and the rate at which these security threats spread. Often the Internet, in fact, can help increase the awareness of security problems and increase the rate at which solutions for known problems can be devised and disseminated.

Security problems come in three basic categories: policy-oriented, network-oriented, and system specific. Policy-oriented problems are studied in detail within information security courses. Network-oriented problems are examined during network security courses. Other system-specific problems are studied throughout system security courses. All of these security problems have two common characteristics: they are caused by people and many traditional attempts to solve them have focused therefore on computer science education.

In computer science education, security continues to increase in importance due to continued growth in computer use and computer intercommunications. Even though existing security problems continue to be solved, unless academia can continue to educate each successive generation of students in security, these successive generations of students likely will replicate the problems their predecessors already have solved. Education cannot prevent security problems altogether, but it may reduce significantly the frequency and magnitude of future problems.

Currently, security problems are considered to be out of hand. Academia desperately needs to take corrective measures now, for the sake of both the immediate and long term future. Industry is already attempting to handle security issues through continuing education efforts in secure coding in addition to more traditional areas of information security, network security, and system security. It is based on this trend in industry that the author proposes an increased focus on secure coding in computer science undergraduate programs.

The author proposes that all computer science undergraduates should begin this increased focus in security immediately after they complete Computer Science I. This would be done through a code-oriented secure software development course. The objectives of this course would be to teach the importance of code security, to instruct in practical coding techniques for making programs more secure, and to provide practice in these secure coding techniques. The proposed course would teach students these secure coding techniques without introducing any significantly more complex data structures or algorithms, leaving those topics for a traditional Computer Science II course.

There are two notable rebuttals to the author's proposal. The first rebuttal is "Computer Science is not programming!" Yet as Dr. Bjarne Stroustrup explains, this may be true at academia's and industry's peril:

> "I find that CS professors often overreact to the inaccurate popular image of the software developer as a lonely guy with 'no life' hacking code all night. To counter, they cry 'Computer Science is not programming!' That's true of course, but that reaction can lead to a serious weakening of programming skills as some adopt the snobbish attitude 'we don't teach programming; we teach computer science' and leave practical software development skills untaught." [67]

Unless academia chooses to require programming competence as a prerequisite for admission into undergraduate computer science programs, programming must continue to be taught. This leads to the second rebuttal, which states that today's undergraduate curriculum already is overloaded. It can be argued that the computer science curriculum does not have room for such an increase in secure programming emphasis without some sort of corresponding decrease. The question of what to decrease in order to offset this increase is outside the scope of this research.

Every software developer, from the initial architects to the maintenance staff, must understand security. Not every software developer has to be an *expert* in security, but they all need to be *competent*. One mistake alone can open up an entire system to serious vulnerability. The access granted by today's Internet and portable data storage devices requires developers to consider any program on any computer system a potential security risk. A program that does not have security in mind from the start of the design should be considered completely unacceptable. Likewise, a buggy program written by a weak coder should be considered just as great of a risk. Both of these problems are addressed by the author's proposal.

An alternative to adding a dedicated security course is to integrate security throughout the computer science curriculum. Some even have suggested integrating security throughout each computer science course rather than inserting it as a short section of each course [36, 45, 69, 75]. Some have shown how this might be done even in Computer Science I and claim to have accomplished this without detracting from the course's other content; however, it also has been shown that Computer Science I already has accumulated many new topics over the years [21]. If new topics continue to be added, it is only a matter of time before Computer Science I will become so overloaded that it must be split into multiple courses or have the individual topics further diluted each time a new topic is added.

A purely-integrated approach, where security is integrated into other courses but no single course is dedicated to security, is not the only way by which security historically has been taught. Academia has taught and continues to teach dedicated security courses to upper classmen and graduate students. The author submits that a purely-integrated approach very well may be a grave mistake. Students may misunderstand, and may believe that security is a topic of lesser importance. A purely-integrated approach also may take time away from the traditional teaching of programming fundamentals at the very beginning of the major.

If a student begins their study of security believing it is of lesser importance, because of its status as a secondary topic in another course rather than being the primary topic of its own course, then the student is less likely to pay security its due attention. Likewise, if a student does not spend enough time focused on basic programming techniques they still may have underdeveloped programming skills when they are required to write more complex software.

Put another way, does one ask a child who still needs training wheels on their bicycle to pull a wheelie?

The author proposes letting students complete Computer Science I before they attempt to cover any significant security content, but immediately after Computer Science I has been completed the student commences a focused study of security in a course on secure programming. The author believes that the students need to be able to practice further their Computer Science I level of programming techniques while learning secure coding techniques that will allow their code to operate correctly, even when being subjected to malicious influences. Furthermore, Computer Science I students do not contemplate "the impossible" as much as they should. Contemplating how to write code that can deal with "the impossible" should become their habit. If students are going to be able to create secure programs, then Bishop and Frincke assert:

> "The final characteristic of robust programming is to assume that impossible events can occur. Impossible cases are rarely that; most often, they are based on an expectation that something in a particular environment will be highly unlikely. Thus, when the environment changes - or a program moves - seemingly impossible events can occur with regularity. This is a difficult concept to teach, in that it requires a broad perspective on possible usage scenarios and a willingness to be penalized, if necessary, to guard against unlikely occurrences. Student programmers must learn to identify the assumptions they do not consciously realize they are making - a paradox. Once they've identified those assumptions, they must learn how far they can go in handling the associated impossibility." [7]

Authors from industry have written books that could be adapted for this sort of code security course despite being written for an experienced, industrial audience [19, 32, 33, 34, 49].

At least for now, the author is unable to require his own students to take a dedicated security course. Such a course would be added to an already overloaded curriculum, which clearly would displace other material if it were to be required. Even if it were to remain a mere elective, it doubtless would displace other electives within an individual student's studies. The author consciously proposes a suboptimal solution to what the author believes to be an intractable problem.

CHAPTER II

REVIEW OF LITERATURE

Section 1: Problem Overview

Computer security problems have existed since the dawn of the digital computer era [39, 57]. Site security - the control over the knowledge of the site and access to the site in which the computer was located - was the principle concern before computers could be accessed remotely and have data electronically transferred from computer to computer. Site security remains an important portion of the overall set of computer security issues but since has been joined by various other issues. Routine remote access to computers has made unauthorized access much easier to achieve. Today, it should be taken for granted that any computer that is accessible from any public network - particularly the Internet - is a likely target for hackers. Routine transfers of data between computers also have facilitated both unauthorized access and the spread of malicious software. Even computers that are isolated from networks are vulnerable to infection by malicious software on portable storage devices. These problems, together with the increasing dependence on computers during both routine and critical activities, have made it necessary to focus additional effort toward combating computer security problems.

Site security and hardware support for security still are valid and important areas of study. They generally are beyond the scope of an academic computer science study of computer security, though. Instead, this study focuses on the computer security problems related to and addressable by computer software. Whether by errors in design and coding or by user-perceived shortcomings in features and usability, all non-trivial software is imperfect. Even trivial software, such as a throwaway developmental prototype program or even a simple “Hello World!” program, can be considered a potential security problem if it either can grant unauthorized access or can deny authorized access due to abuse or even just excessive use [49]. Indeed, all non-trivial computer software is a potential security problem and even some trivial software can be a potential security problem. While careful, skilled, and extensive testing may find some problems, testing rarely ever will locate all problems in a real world program [33, 34, 49, 77].

Software development, like so many other fields of business, operates under management-imposed priorities and constraints. Historically, for many projects, the priorities have been features and time-to-market [35]. Constraints upon budgets and staffing have limited the work on the aspects of projects which management considerers to be of lesser priority. Security too frequently is given such a (lesser) priority. This is because security commonly is analyzed by management using a Return-On-Investment prioritization system [49]. Significant investments in security may not *seem* to yield an obvious return on the investment required to create them unless someone invests even more time and resources by reviewing log records regularly and reporting their findings to management. A better model for setting the priority on security might be to consider security like an insurance policy [49]. Any good business manager knows that a sufficient level of insurance is a necessity. Attempting to operate a business without sufficient insurance may be possible for some period of time; however, when a major problem occurs, insufficient insurance may destroy the company. Likewise, when a major potential computer problem occurs, a sufficient level of security may prevent it from becoming a problem at all whereas even just a single flaw in any critical component may doom the company. If management can be taught to think of security from the perspective of insurance, then security is more likely to get proper attention in the future.

How can management be taught to think of proper security as being like proper insurance? Most managers are university-educated. In the software development industry, most managers were either business management majors or computing majors who moved up into management positions as their careers progressed. One solution would be to reexamine the educational curricula that future managers are going through and adjust them as needed to give those future managers a more proper attitude toward computer security.

The developer's own attitudes frequently both can compound and can explain the problems of management-imposed constraints. Too many developers simply ignore the potential for security problems in their projects [23]. They seem to believe, according to West and others, that security is someone else's problem [17]:

> "[P]eople tend to believe they are less vulnerable to risks than others." [74]

It might seem like security is best left to the experts: those with majors, minors, concentrations, and areas of emphasis in computer security, network security, information assurance, digital forensics, and other similarly specialized areas. Security is not just a problem to be dealt with by the experts, though; security is everyone's problem [26]. The fact is that a single security flaw in a single program can leave not just a single system vulnerable, but potentially open up all other systems on the same network as well. While a small staff of security experts indeed may be a very good investment for any sizable software development project, all of the individuals associated with the project need to have a basic competence with security. This includes the initial architects and designers, the coders, the testers, the deployment experts, and even the maintenance team.

Perhaps it is because of these priorities, constraints, and attitudes that security has been treated as an afterthought in industry for so long [37]. Too often it has been something that would be primarily left to the testers, and even the users, to discover the need for security. Once a security flaw was reported a sufficient number of times to be considered more than just a mere fluke or an odd glitch, the maintenance teams quickly would be instructed to jury rig some sort of patch that would prevent further exploitation of the problem. Unfortunately, all too often the maintenance teams do not have the glamour to attract the best talent. Even if the maintenance teams do understand the system that they are maintaining and the nature of the security problem sufficiently, the problem may not be easy to fix if it actually is due to some of the more fundamental assumptions in the software's architecture and design. Many companies are discovering that it is not as economical to make massive changes late in the project as it is to design the project better and implement it more carefully from the start. This habit of treating security as an afterthought is beginning to change, though. Industry has realized that they need to integrate security throughout their software development processes [53]. However, industry is not the only place that security has been treated as an afterthought. Academia has done it as well [69, 70].

Industry has begun to recognize that academia has not been preparing software developers properly to deal with security issues properly, and industry has begun to take that responsibility partially upon themselves [66]. Fortunately, academia has recognized the problem as well, and knows that its graduates lack the knowledge of security issues that they need [4, 11, 51, 66, 78]. Unfortunately, knowledge of security issues alone is not enough. Too many students are graduating without the skills to deal with security problems, even when they are aware of the problems [78].

Section 2: Historical Academic Approaches to Computer Security

The reason so many students lack a sufficient security knowledge and skill set is due, in part, to the fact that security coursework is not available in many universities [51]. Several universities gravely are lacking in any faculty who have a real interest in security, let alone any faculty who specialize in security [48, 69]. Numerous universities do not have a sufficient number of faculty members to teach security in addition to the other coursework that currently must be included in the curriculum. Neither do many universities have their curriculum structured in such a way to allow their students some time for security coursework to be taken. Universities that do offer security coursework normally do not offer said coursework until late in the undergraduate program or even the graduate level [51, 56, 69]. It is impossible to expect most software developers, many of whom only have an undergraduate degree, to be competent in computer security if security continues to be something generally left for graduate level studies. If security is required at some point in the undergraduate program, then some level of competence can be assumed. Unfortunately, few universities require security coursework at any level.

Too many universities that actually offer security coursework merely offer it as *elective* coursework and do not require either their undergraduate students or their graduate students to take any security coursework [26, 44, 50, 51, 54]. While the availability of elective coursework still may be better than not having the option to take any coursework in security at all, this still results in significant numbers of undergraduate students, and even graduate students, having insufficient exposure to security. In many cases, including that of the author, it is possible for someone to complete computer science degree programs all the way up through - and including - the doctoral level without taking a single course dedicated to computer security. While it should be noted the author has made an extensive study of security as a portion of the doctoral research project and has taught a course on computer security, this does not suggest in any way that the average undergraduate student or graduate student similarly lacking clear indication of security coursework from their transcript will have any similar personal studies of security.

There are three primary approaches or subtopics most frequently covered in an academic study of security: information security, network security, and system security [60]. Sometimes a course on one of these three will be offered as a focused examination of that subtopic of computer security. Other times, two or perhaps even all three of these subtopics will be combined into a single course on security. Occasionally, another approach will be covered during one of these courses, though, that being a code-oriented approach to security, as it relates to the other security topics in that course. Lastly, there is without question a certain level of computer security that can be and is being taught to non-majors [16].

Section 2.1: Information Security

Information security is the study of policies, procedures, and technologies that can be employed to protect the confidentiality [9, 30, 40], availability [40], integrity [9, 30, 40], authenticity [9, 30], and non-repudiation of information [28, 30]. Sometimes known as information assurance [40], this area of security predates even the earliest parts of the history of computing. Information security has been and continues to be a focus of incredible importance both inside and outside of the computer world.

The objective of *information confidentiality* is to ensure that no one can obtain understandable information except those duly authorized to do so. This extends to information in non-volatile storage, information in volatile memory, information in transit between computer systems, and in some cases even to information in transit inside a single computer. Many techniques have been developed to protect the confidentiality of information, but none prove more effective than the prevention of access to the information. If one is not able to access the information remotely, then other techniques are unnecessary; however, breakdowns in access control policies and procedures routinely occur, particularly in transit, and it also is quite common to attempt to conceal the very existence of the information, also known as *steganography*. The use of steganography can be traced back at least 2,400 years in the writings of Herodotus [20], yet information sometimes is discovered despite the best efforts to conceal it; therefore, it also is quite common to attempt to obfuscate the information as well, also known as *cryptography*. The use of cryptography can be traced back at least 2,000 years in the writings about Julius Caesar [38]. It is quite possible that steganography and cryptography date even further back in history than believed, and yet - despite their ancient roots - they remain areas of considerable interest and importance within the computer world.

The objective of *information availability* is to ensure that information can be accessed by all those duly authorized to do so. One of the earliest known techniques for ensuring information availability is the engraving of said information into stone displayed in public places. Unfortunately, this technique is in direct conflict with information confidentiality and so cannot be used with confidential information unless steganography, cryptography, site security, or a combination thereof also is employed. A more modern technique is the duplication and distribution of numerous portable copies of the information. While this duplication and distribution technique has been proven extremely successful with *static* information, including some documents well in excess of 2,000 years old, it clearly and quickly fails as a technique for the storage of *dynamic* information. When the information is of a dynamic nature, there must be a reliable means for determining whether a copy of the information is outdated and updating the outdated copy. An even better approach is to update preemptively all outdated copies without delay once they become outdated. Particularly when dealing with dynamic information or even just copies of static information, there also must be a means to ensure the integrity of the information, something that remains a topic of study even to this day.

The objective of *information integrity* is to ensure that information is not changed except by those duly authorized to do so. One of the earliest known techniques for assuring information integrity was to store the information on some sort of medium that could be written once, but for which any attempt to edit the information would be immediately obvious. Engraving the information in stone or writing the information in permanent (often slightly acidic) inks on papyrus or parchment were some of the most common ancient techniques to ensure information integrity. The boundaries of the information were often marked with either borders or fancy greetings and salutations to prevent any additions from being prepended or appended. Any attempt to alter the information would be obvious immediately and therefore reveal a lack of integrity. This same technique remains important to this day and frequently is referred to in the computer world as a WORM (Write Once, Read Many) storage system [19]. While this system works well for static information, it has difficulties with dynamic information unless it is considered acceptable to create a new WORM record every time the information changes. Despite this, in today's information-oriented world, some places consider WORM records not just acceptable, but actually prudent or even legally required. In the case of dynamic information that is not stored on WORM devices, perhaps due to the frequency and volume of changes, more elaborated measures must be taken to ensure information integrity. In these cases, multiple backups frequently are used to allow comparison between copies in order to detect any failures in information integrity. However, all of this assumes that physical storage of the information is being dealt with exclusively, rather than electronic transmission of that same information. In the comparison of multiple geographically separated and frequently updated backups, this clearly is not the case. The issues involved with the assurance of integrity in electronically transmitted information are substantially different. Error detection codes, error correction codes, and cryptographic systems are each currently in use for assuring the integrity of electronically transmitted information. These systems, as well as systems for updating, comparing, and synchronizing geographically separated backups, all remain modern research topics of significant interest in the computer world.

The objective of *information authenticity* is to ensure that information was produced by those who claim to have produced it. One of the earliest known techniques for assuring information authenticity was to have a mutually trusted agent deliver it from the producer to the consumer. This technique remains important to this day both in the computer world and beyond. Within the computer world, the use of mutually trusted encryption protocols for the protection of communication sessions is very common to ensure that whatever the consumer received did indeed come from the producer. Nonetheless, middle man attacks continue to occur, even against communications sessions protected by widely trusted symmetric encryption systems. Another early technique for assuring information authenticity was the use of a personal mark, personal signature, or personal seal. The required use of personal signatures to finalize financial documents, and to make legal papers official, continues to the present day and can be expected to continue well into the foreseeable future. Likewise, the use of digital signatures has become common within the computer world to ensure the authenticity of digital information both with stored information and with transmitted information. These digital signatures are frequently either an appended duplicate copy of the information or perhaps the only copy of the information in question, in either case, with the digitally signed portion having been encrypted with an asymmetric encryption system where the encrypting key is kept private by the producer but the decrypting key is made public to those who have legitimate need to verify the authenticity of the information. As the level of protection provided by older encryptions degrades, every stored copy either must be updated periodically by the information's producer with stronger, more modern asymmetric encryptions or the level of assurance for the authenticity of the information will decrease over time. This further necessitates continued and, likely, even perpetual research into asymmetric encryption systems.

The objective of *information non-repudiation* is to ensure that information cannot be disavowed by those who produced it. One of the earliest known techniques for assuring information non-repudiation was the use of a receipt. The most common situation for the use of a receipt to ensure information non-repudiation is almost certainly a sales receipt documenting the list of items sold and the prices at which they were sold. In the case of an authentic receipt from a finalized sale, this generally is considered sufficient legal proof of sale and the terms of the sale to prevent the seller from subsequently disavowing the sale or debating the terms of the sale. Another common use of a receipt is the post office's proof of delivery receipt. This commonly is used as legal proof that the post office did indeed deliver the specified packet to the specified recipient and show when and where the delivery occurred. However, unlike the example of a sales receipt, a post office receipt does not always make clear the contents of the packet in question. Within the computer world, log records can provide some assurance of information non-repudiation. If the authenticity and integrity of transmission logs can be assured, then the non-repudiation of information transmission may be assured as well, so long as the logs contain sufficient details about what it was that had been transmitted. Likewise, if the authenticity and integrity of login and activity logs can be assured, then the non-repudiation of information locally generated may be assured as well, if the logs contain sufficient details about the activity in question. From another perspective, though, this area may be considered a special variant of the information authenticity problem if a digitally signed receipt with sufficient detail about, or preferably a complete copy of, the information in question is produced. The critical difference being that the aging of the digital signature would eventually weaken the digital signature's ability to ensure non-repudiation, and therefore research into information non-repudiation likely will continue into the foreseeable future.

Modern academic studies of information security typically include technological topics such as steganography, symmetric and asymmetric cryptography, data backup systems, WORM systems, and digital signatures. They also include even more extensive studies of policies and procedures such as access control, the storage of information in secure geographically disperse locations, layered defenses [42], business continuity and disaster response, and incident management [9]. Given the incredible amounts of data used and generated daily in modern society, the importance of information security cannot be overstated; however, a proper study of information security traditionally requires a database course prerequisite, and perhaps also a file systems course prerequisite, in order to provide the necessary foundation in modern data storage techniques. A networks course prerequisite also is important for the study of information security related to information in transit through computer networks in order to provide the necessary foundation in network hardware technologies and software communications protocols.

Section 2.2: Network Security

Network security is the study of policies, procedures, and technologies that can be employed to protect computer networks and the connected systems from threats originating within the network, in addition to those originating outside the network. Network security's primary objectives include authentication of users attempting to access the network segments and their associated systems, exclusion of malicious software and data from the network, and prevention of snooping or modification of transmissions in the network. Common network security tools include authentication servers, gateway proxy servers, demilitarized zones, network monitors, anomaly detection systems, honey pots, firewalls, antivirus software, and antispyware software [38, 64, 68].

The objective of the authentication of users is very important to insure that only authorized users can access the network, and even then insure that they can access only those systems and network segments that they are authorized to access. Some networks allow users who have been able to log in to a system to access that segment of the network and public resources of the network, including external access to the Internet, even if they logged in to that system using a guest account. These networks are basically public access networks, whether by intent or not, and the systems connected to them should be considered to be at high security risk. Other networks will not allow any users to access the network unless they have been authenticated by a network authentication server, even if they were able to log in successfully to a system on the network. Others will go so far as to exclude any system for accessing the network until said system's MAC address is added to the authentication server's list of systems authorized to access the network. These private networks only are more secure than public networks if the authentication servers and users' authentication information can be kept secure. In the case of connection requests originating from outside of the network, it is not uncommon to require authentication to a proxy server at the network gateway in order to enter the network and access any systems therein. This technique also can be used at the boundaries between segments of a network that have significantly different security levels. It also increasingly is becoming common to set up a *demilitarized zone* to contain network resources that need to be exposed to lower security network segments or to external networks, including the Internet. This demilitarized zone then would have its access to the more secure internal network segment(s) carefully controlled to prevent unauthorized access to the more secure network segment(s) [65]. At the access points between different segments it is very common to have network monitors, sometimes called packed sniffers, and anomaly detection systems in use to detect policy violations as well as monitor technical conditions on the network. When policy violations are detected, they sometimes can be traced back to authorized users attempting to do things which violate network policy settings but actually may be completely acceptable under organizational policy. For example, the author once heard an engineer tell of the difficulties she encountered with an access control system while trying to find technical data on a particular type of "screw" she proposed to use in the machinery she was designing. The author also has personal experience with web pages discovered through Internet searches being analyzed and falsely classified as "pornography" and blocked until the author had a network administrator examine the site, confirm the false positive, and release the block. However, not all policy violations are innocent. Many policy violations are indeed evidence of unauthorized activity, either by authorized users or by intruders that somehow have penetrated the network. Some of these intruders utilize stolen credentials and others use forged credentials but in either case, the ability sometimes to discover them by their policy violations or other anomalous behavior is growing in popularity.

The objective of excluding of malicious software and data from the network is very important for maintaining good technical conditions within a network. Even if they fail to cause any additional harm to the network and its connected systems, viruses, worms, spyware, Trojan Horses, spam, and many other forms of malicious software and data all consume network bandwidth at the very least. If the network traffic containing these malicious programs and data can be intercepted and dropped at either the network's exterior gateway or the boundaries between its segments, then it will minimize the amount of wasted bandwidth and the number of systems exposed in case those systems are vulnerable to that particular threat. Indeed, most systems are vulnerable to denial of service attacks and even more so to distributed denial of service attacks. These attacks can result from the delivery of any sort of network traffic - be it a virus, spam, or even seemingly innocent web page requests in the case of a web server.

The objective of prevention of snooping or modification of transmitted packets is very important for assuring the confidentiality and integrity of information in transit through the network. Encryption is one of the most common techniques to ensure the confidentiality of information against snooping during transit. Symmetric encryption systems using network shared encryption keys may prevent an outside system that has infiltrated the network from compromising the confidentiality of information in transit, but they are virtually useless against snooping perpetrated by a system authorized to share that network encryption key. Private session encryptions are much more secure than shared network encryptions but they presume a private and secure way to negotiate or even prearrange the session encryption keys. Confining transmission packets to the minimal number of network segments necessary to transfer the information from source to destination also is helpful and is one of the primary results of the replacement of older network hubs with more modern switches and routers. Asymmetric encryption also is very useful for assuring information confidentiality and is one of the most common techniques to ensure the integrity of information against modification during transit. Asymmetric encryption also is highly effective for negotiating private session encryption keys in a network where snooping otherwise would be a serious concern.

Reliable network security is a necessity for any network that will be connected to the Internet, or any other public network in any way. A proper study of network security clearly requires a networks course prerequisite in order to provide the necessary foundation in network hardware technologies and software communications protocols.

Section 2.3: System Security

System security is the study of how to protect a computer from both human and automated exploitations. It includes the study of secure operating systems, firewalls, antivirus programs, antispyware programs, anomaly detection systems, log audits, software updating, and - in many cases - even hacking tools and malware attack techniques [65].

The study of operating systems' security features and characteristics is a very important part of system security. Certain operating systems are more secure than others. Some operating systems are designed with security as an important foundational principle. Most of these operating systems achieve their greater security due, in part, to careful isolation of users from other users and processes from other processes. While heavily criticized by some as being very wasteful of resources, this can be a very effective technique and is gaining popularity, as can be seen in the growth of virtual machine usage. Some operating systems also can be made considerably more or less secure, depending on how they are configured. Access permissions in particular have a huge potential impact upon a system's security. The permissions set on individual files and folders can be customized to limit which users and applications may access them; these should be set so that the smallest number of users and applications are allowed access, while still granting access to all those who have legitimate need. Types of access also can be customized, so that only those particular permissions that individuals and applications actually need are granted to them. Additionally, overall permission of multiple accounts can be categorized into groups and then customized either to authorize or to limit their activities and their access permissions on a system-wide basis. Proper use of operating system provided permissions can make significant improvements in security on a system-wide basis. Conversely, improper use of operating system-provided permissions can create significant vulnerabilities in security on a system-wide basis. The security of the operating system is one of the most fundamental issues in overall system security.

The study of hardware and software firewalls is a very important part of system security. Firewalls are used for blocking unauthorized transmission into and out of a computer. By limiting the transmission ports and protocols to a certain authorized list, many potential problems will be prevented completely before they can ever occur. Also, more modern firewalls can be tied in with antivirus software and antispyware software to scan all incoming packets for malicious software and all outgoing packets for communications to known botnet and spyware controllers. Anomaly detection systems also can be configured to scan those packets attempting to pass though the firewall in either direction for policy violations to provide even greater defense in depth for the system at the firewall border with the network.

The study of how to audit system logs is a very important part of system security. Frequent audits of the system logs also can reveal evidence that the system has become a target of interest to the outside even when it has not yet successfully been penetrated. If usernames for unknown users begin showing up in the logs, or if unsuccessful login attempts show up and are not immediately followed by successful logins from those same accounts, even very slow and sporadic penetration attempts may be uncovered. Furthermore, even successful logins from an authorized user may reveal unauthorized access, particularly if the logged connection source IP addresses or times do not conform to established patterns for the user in question.

The study of software update issues is a very important part of system security. Many modern software systems are capable of automatically updating themselves, some even when they do not have a user currently logged in. This frequently is accomplished by having the software activate and connect to an update server every so often to inquire if there are updates available. While many users may enjoy not having to invoke manually their update routines for their systems, these automatic update features actually can be both good and bad, depending on how the system is being used, how the update features are configured, how critical the software is to the system in question, and how well the updates are tested prior to release. Some users will not remain logged in to their system when they are not currently using their system. If the automatic update features

- and perhaps even the operating system itself - are not configured to allow updates while the user is not logged in, it is likely that update routines may be triggered upon user login. If the user is logging in to perform some time-critical work, it may be very inconvenient for any number of update routines even to begin checking for the need to update let alone actually downloading and installing updates each time the user logs in. Users that have had too many bad experiences with such frequently will find ways to disable the automatic update systems, lower the frequency of checking for updates, or leave themselves logged in to their system while they are not using the terminal and potentially even while they are away from the terminal. These responses can result in lower security as updates get delayed and can allow unattended and unlocked terminals to be accessible to other individuals. For critical software, substantial delays in installing updates unacceptably may extend windows of opportunity for system exploitation, even after solutions to the problems are well known and widely available. Unfortunately, not all software updates are as fully tested as they should be. A recent example of such was when an antivirus manufacturer released an update which falsely identified and quarantined the critical Windows XP Service Pack 3 svchost.exe file as malware. This caused all such affected systems to crash as soon as they were rebooted [46]. Additional reboots would not solve their crash-on-reboot problem. Computer support technicians were forced to visit and manually restore each affected machine including at some rather large companies that experienced failure in the majority of their personal computer systems [47]. This, and other similar problems with software updates from many different vendors, has led some to question if one should install updates as soon as they are available or if it might be better to prolong the potential risk of exploitation in order to allow other earlier adopters to test more fully the updates.

One of the most controversial historical aspects of system security is a movement towards teaching more students how to hack and how to write malware [18]. It has been pointed out, though, that the best defenses are the ones that are constructed by those who must fully understand the attack techniques that may be employed against those defenses [6, 44, 49]. While this historically has been done in some academic settings, it has not yet achieved universal acceptance in academia and is met with particularly strong resistance in the administration, the university's lawyers, and in the information technology support department at many universities [71].

Without reliable system security, all the best efforts toward attaining information security and network security are for naught. A proper course in system security requires at least an operating systems course prerequisite in order to provide the necessary foundation in among other things, process scheduling, process management, and inter-process communications.

Section 2.4: Code Security

Code security is the study of well-known coding techniques that have significant security implications. Code security topics often include information leakage, failure to protect data and code, numeric overflow and underflow, buffer overflow, exception and error handling, race conditions, environmental assumptions, issues of trust, usability issues, unnecessarily high execution privilege, command injection, SQL injection, string format attacks, pseudorandom rather than truly random number generation, and weak cryptography [19, 33, 34]. Some of these issues are discussed as general design and coding issues in courses such as Computer Science I, Computer Science II, and Software Engineering. Others are discussed less frequently as generic design and coding issues in any other courses and are therefore sometimes never discussed at those universities that do not teach code security coursework.

Information leakage historically has provided huge amounts of information to unauthorized recipients and, in many cases, lead to even more serious problems, both inside and outside of the computer world. Information leakage can happen any time a program provides a user any information other than that which is absolutely necessary for the user to accomplish their authorized work. Information leakage also can occur when those charged with protecting the information incorrectly have authorized individuals to access information they have no need to access, but that is more in the domain of information security than that of code security. Information leakage also can occur when computer storage devices are read by users that should not have the ability to read them, but that problem is more in the domains of information security and system security than that of code security. Rather, the information leakage problem within the domain of code security involves the program being designed and coded to provide information to the user that it does not need to provide, including such things as displaying unmasked passwords during login.

At first, the failure to protect data and code may sound as though it was related to information leakage very closely. Actually, this is not the case. In the realm of code security, protection of data deals with making data private to pieces of the same program - and to other programs which share part of the owning program’s memory - that does not need to access that particular piece of data. Protection of code, in the realm of code security, deals with making the code immutable during its execution, and potentially making it unreadable to other processes outside of the operating system and security applications.

Numeric overflow and underflow normally are assumed simply not to happen in their programs by far too many programmers. In many cases the operands in question might be constrained to prevent overflow or underflow from being possible in some way. However, in other situations, when the numeric result both will be output immediately to the user and when any overflow or underflow would be obvious immediately, it might not be necessary to check for overflow or underflow. If the numeric result either is stored or used in future calculations, then the code needs to check for overflow and underflow. Fortunately, overflow and underflow are very easy to detect at the hardware level. Unfortunately, if they are not detected and the erroneous result is then used in further calculations, it potentially can produce disastrous results in the case of critical code. Even in less critical code, numeric overflow or underflow can lead to other code failures, most obviously including buffer overflows, command injection, and SQL injection.

Buffer overflow is a very common problem in languages that do not automatically track the size of an array and check to make sure accesses to the array are within the bounds of the array. C language - one of the common languages in use today - along with several derived and otherwise related languages, is susceptible to buffer overflows. Proponents of these languages rightly will point out that susceptibility does not guarantee problems and that properly designed and coded programs in these languages can be completely free of any buffer overflow problems; some will go so far as to point out that the very lack of bounds checking should be considered advantageous for the sake of execution speed in that the time for bounds checking will not be wasted in those situations where the indexes somehow are constrained always to fall within the array's bounds. Others will point out that it is a simple process to design and code a compound structure and a set of routines to operate on it that will perform bounds checking if the overhead for such is considered acceptable. Furthermore, in the case of inlined and optimized subroutines, the overhead actually may be partially if not completely optimized back out of the resultant code if the optimizer detects the code to be unnecessary.

Exceptions and errors are sometimes checked for in particular code routines and reported back to the calling code for that code to deal with appropriately. Unfortunately, particularly when a different coder is writing the calling code, not all calling code examines the returned value to determine if an exception or error occurred. In these situations, false assumptions about the successful completion of the called routine may cause additional exceptions and errors to propagate further through the code. This potentially can lead to some things as seemingly benign, for example, as the crash of a particular module of code. However, when this is a critical module, such as a security subroutine, the results truly can be disastrous. Moreover, exception and error propagation potentially can lead to the exploitation of any other type of security flaw.

Race conditions are some of the most difficult of all security flaws to detect strictly through testing. They can be difficult to test deliberately for due to the timing issues involved in getting the exact interleaving needed to cause the race to produce obviously erroneous results. When race conditions do occur, the resulting impact can be potentially insignificant, potentially severe, or even cause human fatalities, as it was in the case of Therac-25 [52]. Potential race situations should be easy to find by manual inspection of source code when there are a very small number of different thread types, but as the number of CPU cores and pipelines continue to increase, the number of threads in modern programs are similarly likely to increase as well. In a program with a large number of different thread types, detection of some potential race conditions still should be possible with compiler assistance.

Assumptions about the software environment in which a program may operate can be another difficult potential security flaw to detect strictly through testing. Depending on how paths are ordered, what the directories in the paths contain, and what is in the current working directory, a program may work exactly as intended during all testing; however, with slightly different environmental conditions, the same program may open up a host of potential security flaws. While most software developers may think of their code as being relatively or even completely self-contained, any code that must make system calls is not truly self-contained. Any calls to other pieces of code that are not defined fully and statically within the calling code have the potential to be exploited, either intentionally or accidentally, if other pieces of code by the same name become substituted for the code that was intended to be called. Even simple version changes can produce unexpected results, varying from an incorrect result or runtime crash on the mild extreme to a full-blown security breach in the worst case.

Even when the software environment is indeed as expected, there still may be issues of trust that should be considered when interfacing with third-party software. Many large projects teams license and include some third-party software as a portion of their project in order to reduce costs and to protect themselves from accusations of intellectual property theft. When such inclusions are not integrated fully into the resultant project package, but instead deployed in parallel, risks can be introduced by future updates to the third-party software.

Usability issues range from potentially insignificant to potentially severe. Prospectively the most serious of the security implications of usability issues are when the difficulty using security features and settings alienate users and administrators from using the software's available security to its utmost. Fortunately, usability issues related to security should be detectable during extended end-user testing by observing the level of security usage by the users during the tests.

Some programs execute with unnecessarily high levels of privilege. While some programs indeed do need extensive privilege to operate correctly, few require full root / system administrator privilege. All programs should have their proper execution privilege requirements determined and documented at design time. This should be re-verified during the testing and deployment stages of their development, and their install routines should be customized and documented accordingly. Furthermore, some programs actually can be partitioned into different modules that require different levels of permissions, allowing each module to have its permissions set as low as possible, and thereby further reducing their risks. In the ultimate example of problems that can occur in this area, some users - either knowingly or not - do all of their work in a root / system administrator privileged account; thereby allowing all of the programs they execute to inherit their root / system administrator level of privilege, even if the programs themselves are able to run successfully with lower user privilege levels.

Command injection is one of the most hazardous categories of security flaws, due to its potential to allow the complete takeover of the compromised system. Buffer overflows are a notorious mechanism for command injection, but routines for allowing commands to be passed to a shell or even the operating system itself for execution also are potential causes for command injection. One of the most subtle mechanisms for command injection is the manipulation of the execution environment, particularly the current working directory and possibly even the path, in such a way as to substitute a Trojan Horse for a called external program or library routine. Once a technique for command injection is detected, a common tactic is then to open an independent terminal window (“xterm &” on most Unix-like systems) or file system window (“explorer” on modern Windows systems) for the user to inject even more commands to run using the permissions level of the compromised program. Whatever the cause, command injection can be prevented though environmental sanitation and input data validation.

SQL injection, a variation on command injection, has been the plague of database systems for years. Many different database front end systems do not validate their inputs properly before passing them on into the code that does the actual database accesses. This can compromise both the confidentiality and the integrity of the information throughout said database.

String format attacks are another variation on command injection. If the code does not validate variables passed in to control the format of string output operations, malicious format codes can leak information and at least in FORTRAN, even order the shutdown the entire system. Fortunately, string format attacks easily are avoidable if the coders are willing to code a more complex sequence of operations that determines what format is needed and then outputs the string with the appropriate formatting already hardcoded into the instruction.

Most developers have heard at some point or another that most computer-based random number generators are actually only pseudorandom. These generators are good enough to fool the casual human observer into thinking that they are random sequences but can expose the internal state of their generator if their algorithm is known and if a sufficient number of generated values can be observed directly, or sometimes even just observed indirectly. While these generators are useful and sufficient for entertainment and random simulation programs, they can never be depended upon for anything that involves a true need for secrecy. Instead, cryptographically strong random number generators and true entropy generators should be used when there is a true need for secrecy - when there is a need to be sure that the next number can never be predicted based on any prior observations.

Modern cryptography is based heavily in mathematics, and in many cases is dependent on having access to a random number series of significant length. Cryptographically strong random number generators and entropy generators are capable of generating these series of numbers, as are other sources, but modern cryptographic systems are not invulnerable just because they use cryptographically strong random number generators or entropy generators. Modern brute force decryption techniques continue to advance in capability, reducing the strength of previously secure algorithms and key sizes. Algorithms must be re-examined continually to insure their continued validity in this ever-changing security environment. Furthermore, it is not just the programs that currently are in use that need to be updated periodically, but also the encryptions of data in long term storage. For example, it is very realistic to expect that medical records for individuals born in the present should be kept confidential for more than a century given the state of modern medicine. Most encryption systems in use today certainly will break down before that century is over, requiring the records to be either decrypted and re-encrypted with a better system or super-encrypted with a better system. If the records are super-encrypted, then access times will suffer for all future read or write operations that will therefore require multiple encrypts or multiple decrypts. On the other hand, if the records are instead to be decrypted fully and then re-encrypted fully, then there is the potential for information leakage during that process. While modern encryption systems definitely can reduce the likelihood of information leakage, except for those that have been proven immune to even brute force quantum attacks, perhaps it should be assumed that modern encryption systems do not guarantee success but merely attempt to reduce the likelihood of information leakage.

Various computer security courses have started to include some coverage of code security as a portion of the course and as it relates to the information security, network security, and / or system security topics that the course focuses upon. An information security course *might* discuss SQL injection attacks. A network security course *might* discuss which random number generators are cryptographically strong and which are not. A systems security course *might* talk about buffer overflow attacks and command injection attacks. These tend to be very small portions of those courses, though. They also tend not to be covered till late in those courses after substantial amounts of time have been devoted to information security, network security, or system security theory. As a result, the students may tend to focus on the information security, network security, or system security theory as being the most important security topic in the course. They may consider code security to be of lesser importance, as it is appears to them as not being important enough to attain its own course.

To date, the author has not been able to find a single academic course, other than his own course, dedicated primarily - let alone exclusively - to code security. Industry, on the other hand, seems to be offering more and more code security continuing education courses for the college graduates in the industrial workforce.

A course on code security is unique among the areas of computer security studied by computer science majors in that the only prerequisite needed before such a course would be Computer Science I.

Section 2.5: Computer Security for Non-Majors

Lastly, there is a certain amount of computer security that rightly is being taught to non-majors. Frankly, there are certain things that any modern computer user needs to know about computer security. Despite their considerable use of technology prior to college, the typical college freshman does not know what they need to know about computer security. Some universities offer computer literacy courses to non-computer science majors which cover these topics, among others. Some computer science majors will take these computer literacy courses, particularly if they do not have an extensive prior technological background, but those computer science majors that do not take a computer literacy course still need to cover the same computer security for non-majors topics, perhaps even more than the non-majors themselves need to.

Some of the most important concepts that all computer users need to understand deal with the creation, storage, updating, and reuse of usernames and passwords. Both usernames and passwords should be created to be as unpredictable as possible if the user has any say in their creation. The sole exception to this rule is in the predictably of usernames that are publicly released as a form of contact information, such as email addresses and web homepage addresses. Despite the need to have usernames and passwords as unpredictable as possible, they still need to be easy enough to remember that the user does not need to store them elsewhere. Passwords need to be updated frequently enough to limit the window for exploitation in case the password is compromised but not so frequently as to make them difficult to remember. Unpredictable usernames never should be reused from account to account and passwords should never be reused from account to account or from time to time. Likewise, trivial variations upon unpredictable usernames and passwords from account to account and from time to time never should be allowed. Usernames and passwords also never should be shared between different individuals. While these rules will not guarantee that username / password pairs cannot be broken, when coupled with automatic lockouts after a certain number of failed login attempts, these rules can reduce the likelihood that username / password pairs will be broken very quickly.

Limited access accounts should be used for most computer work and administrative privileged accounts should be used only when absolutely necessary. Antivirus software, antispyware software, and firewalls should be used in order to minimize the risk of exploitation and hard drive cleanup utilities should be used to minimize the damage that can be caused by successful exploitation. All confidential information should be kept both encrypted and backed up elsewhere further to minimize the damage that can be caused by successful exploitation. Publicly accessible computers also should be rebooted both before and after use to reduce the chance that Trojan logins might be running before use and that keyloggers might have left information in memory after use.

File and directory permissions should be set to restrict access as much as possible and shared folders should be used as little as possible. File sharing programs should be restricted as to the folders they can share, if file sharing is allowed at all. Users also must understand the legal environments in which they will be operating and that if their legal environments change, that they need to become aware of their new legal environments.

Individuals also should attempt to limit the personal information they deliberately make available via social networking sites to reduce the amount of information available for social engineering attacks. They also should know not to give out any personal information or account information through communications that they did not initiate personally to avoid phishing via email and forged websites. In fact, the transmission of any personal information - even to trusted and verified destinations - should be avoided from publicly accessible computers and over wireless networks if at all possible to avoid snooping, unless it is through the use of encrypted sessions.

Lastly, even non-majors need to understand the importance of security being designed into processes from the start. This principle actually goes beyond just the computer world and so is of benefit to more than just those who work in areas related to the computer industries. Regardless, some non-majors someday will be called upon to act as subject matter experts for software development projects, and these non-majors above all must understand that computer security must be incorporated into the computer software development process and designed into the software from the very start.

Computer security for non-majors requires Computer Literacy as a prerequisite or corequisite and frequently is taught as a portion of modern Computer Literacy courses at those universities that offer such courses.

Section 3: Industry's Buggy Code Theory

Industry is starting to become aware of a single commonality between all software security flaws, that they are implemented as code. Some authors, including Ranum, are therefore pointing out that insecure code can be correctly viewed as simply being low quality, buggy code [50, 56, 58].

> "[W]hy are we still treating security as a separate problem from code quality? Insecure code is just buggy code!" [58]

Information security, network security, and system security courses may provide students with some practice writing code, but as they do not *focus* on code, are not likely to improve the quality of their students' code significantly. Even if graduates are experts at information security theory, network security theory, and system security theory, they still may replicate well known coding errors that lead to security flaws if they are never taught what those coding errors happen to be. Indeed some categories of errors, such as the buffer overflow, have been well understood for decades yet continue to be replicated in modern software thereby causing more computer security problems.

While there is some momentum building in academia to view security faults as design failures and increase the emphasis on robust and secure design processes in software engineering classes, it is industry that seems to discuss this buggy code theory the most, even if it is also industry's attempts to cut corners to reduce costs that helps perpetuate the creation of more bugs. Still, the point remains that coders should know better. After all, when a designer specifies that an input should be received for, say a phone number, a date, or a time, why should the designer have to specify to the coder to make sure the buffer does not overflow and to make sure to validate the received information? The coders should already know to protect against buffer overflows, and should not have to have that level of detail appended to each and every single input request given them by the designers! The coders may need to request information from the designers about what localization rules to follow to validate phone numbers, dates, and times, since those vary from place to place. Coders should not have to have every input request they receive specify that the input needs to be validated, though; they should know to do that!

Section 3.1: Security-Aware Compilers to Combat Security Bugs

One idea to help reduce the problem of security bugs is to treat the security problem in the same way the optimization problem was handled in the past: ignore the coders and turn the problem over to the computers themselves. This proposed solution would have developers rely on security-aware compilers and post-processors [58] to fix algorithmic and coding mistakes in similar ways to how optimizing compilers fix inefficiencies in source code [29, 43, 58]. This is being considered because computer software is more methodical and more consistent than humans are. On the other hand, this idea has been criticized as computer software is not perfect and since compiler warnings are often ignored by developers.

Software is very good at automating monotonous tasks and performing very precise work in an especially methodical manner. These characteristics make software an appealing choice for "looking over the shoulder" of coders to make sure that their code is as secure as possible. Where a coder might remember to put certain checks and safeguards into their code most of the time, software is quite capable of finding and pointing out the times when they forgot to do so.

Software can be far more consistent than humans. If a development team is striving to use the same standards, then they may produce code that generally is consistent; however, there still will be variations within what different developers produce. When developers all are using the same version of the same security-aware compilers and post-processors, the resulting level of security should be far more consistent than if the security was being handled by humans alone.

Unfortunately, software is not a perfect solution for the security problem. There are some security problems that software will never be able perfectly to solve by itself [29, 58]. In particular, input validation will always require a human to specify what the rules should be for validating the data. Some argue that many of these sorts of problems easily are solved by simply returning warning messages to the developer at compile time indicating the security problems that had been detected that could not be handled by the software and therefore required the developer's intervention.

Software issued warnings cannot be considered a fully acceptable solution either. There are too many documented instances where warnings are suppressed by default. If a developer has to do extra work just to get extra criticism of said work - even simply that of a compiler or post-processor - then too many developers will not put in the extra work, assuming they even know how to invoke the issuance of the maximum possible list of warnings. Few developers use well known tools that have in some cases already existed for decades, such as lint, that originally were designed as general compilation and post-processing aids, rather than specifically as security compilation and post-processing aids. Nonetheless, such tools can find errors in code that could be exploitable security flaws. If developers will not use tools such as lint, why should anyone assume that those same developers will use security-oriented compilation and post-processing aids? Some developers have even gone so far as deliberately to take action to suppress not just compile time warnings, but even deliberately suppress runtime warnings about potential security flaws rather than correct the flaws.

Section 3.2: Better Education to Combat Security Bugs

Better security-aware compilers and post-processors would be very useful even if they were imperfect, since software developers are imperfect and do make mistakes [37]. These tools would be useful if more developers actually used them, assuming of course that the developers do not simply ignore the programs' warning messages [58]. Indeed, some tools that can reveal potential security flaws, such as lint for example, have existed for years but are not universally used [58].

> "[A]cademia and industry are going after symptoms, (teaching attack techniques, security technologies etc) and not after the cause of the symptoms: In-correct [sic] software." [56]

While security experts do need to be very knowledgeable of attack techniques, security technologies, and many other related topics, Pothamsetty suggests that not every developer needs to know everything related to these topics. Some security technologies, such as security-aware compilers and post-processors, should be a required part of every undergraduate computer science curriculum, but of the historical academic computer science approaches to computer security, the one that best fits with is code security. Even then, it best fits in with a discussion of compilation and debugging standards in the pursuit of quality code. Indeed, Taylor, et al., assert that computer science students need to understand the importance of code quality in the quest for code security.

> "It is imperative to teach students that safe and reliable programs are inherently more secure." [70]

Then, and only then, are they likely to appreciate fully the value of the tools they already have and make proper use of them. Furthermore, history is full of examples of times when the extensive use of particular tools led to further and faster innovation in the quality and effectiveness of those very tools. Therefore, if the computer world is going to rely on security-aware compilers and post-processors, then it is going to have to educate its workforce to use those tools that are already available.

Knowing that academia is in no position to fix the educational deficiencies of those already in the workforce, industry is increasing its own efforts through continuing education courses and training [35, 66]. While some of these efforts focus on information security, network security, and system security, many of them also - or even instead - are focusing on code security, as Ingham points out:

> "Industry programmers are often unaware of the importance of design of secure systems, how attackers exploit programs, and how easy many exploits are. Over the last few years, the focus in industry has grown to include security; this change in focus translates into a need for programmers who can design and write more secure programs. This need results in a demand for continuing education classes on programming securely." [35]

Security experts in industry also are writing books about code security [19, 32, 33, 34, 49]. These are not textbooks for academic use, but rather they are books written for experienced industrial developers [24, 32, 33, 34, 49]. These books have potential to improve the secure coding skills of those already out in industry, but they are not written at the appropriate level for the traditional undergraduate computer science major, and therefore are not ideal to address the current deficiencies in academia as they currently stand. However, these books could be adapted for use even in the freshman year of a computer science degree program if there were a sufficient number of universities interested in having textbooks about code security.

Section 4: Current Theories on Security in the Computer Science Curriculum

Given the magnitude of the security problem and given the level of outcry over it in industry, the consumer base, and even in governmental regulatory organizations, academia is discussing what can be done to address the problem. Full solutions, while desirable, currently are not seen as being achievable quickly or perhaps as being achievable ever. Academia therefore is searching for any partial solutions that may be worth trying and evaluating. Four theories are prominent at this time. One theory holds that security should be a required portion of the core computer science curriculum. Another theory is that security studies need to start earlier in the curriculum. A third and rather radical theory is that security should be integrated into each and every single course in the computer science curriculum. A fourth theory, perhaps the most controversial of all, is that ethical means of teaching hacking skills need to be promoted to provide the same types and the same level of skills to those that defend computers from attack as the attackers themselves posses.

Section 4.1: Security Required in the Core Curriculum

Academia is beginning to realize that security needs to be a required part of the core computer science curriculum [26, 36, 44, 45, 48, 54, 66, 69, 70, 76, 78, 79]. While most computer science majors likely will not specialize in security, academia is beginning to accept that all computer science majors need to have a certain level of security competency. As promising as that might sound, there still is not a consensus that code security needs to be required, as it ought to be. After all, it only takes one bug in one line of code in one program potentially to open up not just one system, but every system on every network that is running that program.

While security, and code security even more so, might sound like unnecessary additions to some traditionalists, it should be recognized that at earlier points in the history of computer science, the typical undergraduate study of computer science did not include some topics that are now required in the typical undergraduate degree. Not just “new” topics like object-oriented programming have been added to the required curriculum, but there was even a time when topics such as operating systems and networks were not ever looked at until one reached graduate studies, yet now they are accepted as part of the core undergraduate computer science curriculum. The time is overdue for security, and even code security, to be of the required core curriculum as well. As Bishop and Frinke assert:

> “The ability to write secure code should be as fundamental to a university computer science undergraduate as basic literacy.” [8]

So since, as it has been stated, insecure code simply is incorrect or buggy code, does that mean that academia is graduating incompetent coders? Even if not intentionally, this is exactly what is happening. The vast majority of the computer science majors graduating today may be minimally competent coders but are not fully competent coders. This situation cannot be permitted to continue.

It most likely will be years before most universities require security as a portion of their core computer science curriculum, let alone code security. Even then, it still will take over a decade to graduate a significant enough number of these code security trained graduates to make up a considerable percentage of active industrial coders. Worse yet, even then there still will be billions of lines of insecure legacy code in use, due to the high cost of rewriting it. Further, some universities likely will continue not to require any security, and their graduates still will continue to perpetuate the production of buggy code so long as they code, unless industry corrects the deficiencies in their education. Why should industry be required to do so, though? Stamat and Humphries explain this further:

> "A growing trend among businesses is to send their inexperienced programmers to security "boot camps". These boot camps provide an accelerated instruction medium for software developers to receive security training and certification. Although this is a commendable response it only sidesteps the issue. Companies have lowered their expectations of today's college graduates and many now require all new hires to go through this additional training. This training only delivers a greater financial burden, both for the company and the consumer. If the development community wants to combat this problem then it's going to have to start from the beginning, in colleges and universities." [66]

Who is better at education, industry or an educational institution? Indeed, it is the colleges and universities that must step up and insure that their graduates are competent to design and code secure software. That only can be done when academia, as a whole, requires not just security coursework but specifically code security coursework of all computer science students.

Section 4.2: Security Early in the Curriculum

Academia is beginning to realize that security, particularly code security, must be started earlier in the curriculum [36, 44, 45, 54, 69]. Students are getting too much practice writing copious amounts of insecure code before they get to upper level security courses that *might* talk about code security. Once they do cover code security, if current curriculum even does, they will have years of bad coding habits to break and not have as much time left before graduation to break those habits as they had to make those bad habits. Students would have less time to develop their bad habits and would have both more time and more academic guidance to help break these bad habits if code security were covered earlier in the undergraduate computer science curriculum. Taylor and Azadegan claim an early start is a necessity:

> "[E]ducation must infuse secure coding and design principles early" [69]

Unfortunately, the current model for teaching code security - as a portion of information security, network security, or system security - cannot be moved too much earlier in the curriculum due to the prerequisites for those other security topics. Perhaps code security needs to be separated from other security topics that must, due to their prerequisites, remain later in the curriculum. Perhaps code security needs to be moved into courses earlier in the curriculum or even made a course of its own. The question then becomes how early code security can be offered and how early does it need to be offered.

> "It is the responsibility of universities to teach future computing professionals secure and robust coding and design principles from the start." [70]

Taylor and Azadegan suggest that academia need to start teaching code security from the very first semester that an undergraduate student is taking any computer science coursework. While certain authors claim to have even integrated some security into Computer Science 0 and Computer Science I, a true code security course is not likely to be viable until after the completion of Computer Science I. Further, without a course of its own, code security may not be understood properly and respected by many students as the topic of importance that it truly is.

> "Software security starts when learning the programming language." [45]

When, though, are students considered by Marks and Stinson to be "learning the programming language"? The programming skills of students that have yet to complete Computer Science I may not mature enough for any discussion of security more than computer security for non-majors. When the students are taking Computer Science II, they are starting to learn about linked lists, stacks, and queues which is in turn continuing to help them with "learning the programming language" and the full depths thereof. They have begun to move on from the very basics of how to program and, though they still are learning more about how to program, they also have begun to learn about why we program the way we program. They are beginning to learn to seriously compare and contrast data structures and algorithms and should therefore be ready to learn to compare and contrast algorithms and code from a security perspective as well. It is during this stage of their education that a study of code security becomes viable and could begin: after the completion of Computer Science I.

Section 4.3: Security Throughout the Curriculum

Perhaps one of the more radical recent ideas being considered to improve computer security is to integrate security throughout the entire computer science curriculum [26, 48, 54, 69, 70, 76]. Each and every single course throughout the entire computer science curriculum would incorporate some aspect of security into the course. Therefore, all together the required and elective computer science courses would cover all of the security topics that the student needed. This would eliminate the need to have any introductory level or required courses dedicated to security; nonetheless, advanced courses that specialize in information security, network security, or system security likely still would exist at some universities in order to allow some students to focus on security for their emphasis. It has been further suggested that security should be integrated throughout each of the individual courses that are focused primarily on other subtopics of computer science, rather than just inserting a security module into the course the same way a course might be inserted into a curriculum [36, 45, 69, 75]. Some universities have even integrated security into Computer Science 0, Computer Science I, and Computer Science II [69, 72]. These integration theories may help significantly to solve the security problem but they present problems of their own.

Students are more likely to recognize and understand the importance of areas of emphasis within computer science when those areas have at least one course dedicated to them. ‘If the faculty does not think this is important enough to deserve a required course, why should I consider this important?’ might be exactly what some students would think, should academia attempt to teach security exclusively in an integrated fashion. It also creates potential scenarios for gaps in some students’ security education in the not uncommon event that some students transfer from one university to another in the middle of their undergraduate programs. Will two universities that appear to have similar courses required in their computer science undergraduate degree programs distribute the various security topics the same way throughout their courses? For some topics like network security, one likely could assume so; however, but for other topics like buffer overflow and command injection attacks, for example, probably not.

If security is to be integrated into all courses in a computer science curriculum, it logically also would impose a requirement for some level of security expertise upon all of the faculty as well as time constraints upon all computer science courses. Not all faculty members are security experts or are even interested in security [48, 69]. Many faculty members also may argue correctly that their course does not have time to do justice to its primary topic, let alone to both the primary topic and a secondary focus upon security [36, 48]. Even if officially directed to include security within their course, some faculty would even go so far as to claim ‘academic freedom’ and then teach only a token amount of security to satisfy the directive. In the end, it would be up to the initiative of the individual faculty member to decide whether or not to give security its due emphasis. It may be possible to overcome some of these problems by integrating bits of security into some courses and not into others. Supposing that all of these new problems could be overcome, though, what then should be done about proposals to integrate programming language concepts throughout the curriculum [5]? Proposals also have been made to integrate software testing [1], team projects [61], ethics [63], robocode [10], HyperCard [31], iPhone [25], and many other topics throughout the computer science curriculum. Curriculum designers will have to weigh the value of these various special topics that might be suggested for integration into and throughout the computer science curriculum as they clearly cannot be implemented.

Section 4.4: Ethical Hacking Courses

Likely the most controversial idea currently being discussed regarding security education is that of ethically teaching hacking skills to undergraduate computer science students [18]. While the ideas of practicing attack and defense strategies have been accepted in military organizations for thousands of years, those same ideas have not come into open acceptance with regard to civilians being taught how to attack computers in academic settings, particularly those as young as the traditional undergraduate student. Indeed, many claim that a large percentage of the lapses in computer ethics actually are perpetrated by those same students who may now be able to obtain a state-of-the-art education in hacking techniques by world-class experts in computer security. Understandably, this idea has many academicians, their administrative superiors, and their university's information technology support department more than a bit uncomfortable.

Despite these concerns, such classes have been taught to upper level undergraduate students and graduate students for over a decade. This type of class frequently is given much more supervision than the typical computer science courses taught in the same departments. Students enrolled in this manner of course frequently are required to sign very restrictive code of conduct agreements before they can participate in the course. This sort of course also normally is forced to confine its hands-on exercises and experimentation to isolated test environment systems and networks [12, 27].

One of the few things that appears to be universal in regards to ethical hacking courses is that the course must cover ethical and legal issues related to hacking [73]. Once an ethical basis has been constructed to justify the teaching of hacking skills, it must be constrained carefully within the applicable legal boundaries to insure that the course and its students do not stray accidentally beyond what is permitted into areas that would require the course to be shutdown and possibly even lead to the prosecution of its participants.

Once the ethical and legal foundation is well established, then the course can move on to simple observation of the environment in which the students are being allowed to work. Simple identification of other machines on the same network segment quickly leads to network traffic snooping, particularly - but not exclusively - of clear text packets. Even encrypted packets can still provide information about the existence of and associations of machines in addition to the types of protocols they are using. In some cases, the types of protocols in use may provide further information about what operating systems and applications are in use on those machines.

Once something is known about what operating systems and applications are available to target, it is common to begin reviewing the well known list of exploits that historically have existed in those operating systems and applications. While any good system administrator will keep their systems updated to avoid patched historical vulnerabilities, it is very unfortunate - but also very true - that not all systems have good system administrators. Many operating systems and applications still may be found to use well known default passwords or may be found to have unsecured mechanisms for resetting the root / administrative passwords.

There are numerous automated attack tools that also can be employed within an ethical hacking course to aid in the detection of exploitable security flaws actually while reducing the amount of detail that the students need to be taught about how to break into various systems. In the terminology of the hacking world, this is the difference between a true hacker and mere script kiddies. While the script kiddies generally are not given the same respect that a true hacker might be given within that culture, the script kiddies are often just as capable for conducting quality penetration testing of a system or a piece of software as a true hacker. The increased training of "script kiddies" within academia would provide an increased population of skilled penetration testers for industry to test more thoroughly future software products.

Some of the easiest forms of automated attacks simply are repeated login attempts against both known and guessed usernames. Such attacks frequently start with dictionary attacks against the password but then may move on to a full rainbow table attack, particularly when done against a known username of interest. Such attacks easily should be detectable by modern network DOS/DDOS protection systems due to the extreme repetition of network login requests, and therefore should not be as effective a form of attack as it is. However, as previously stated, not all systems have the quality of system administration that they truly need.

For ethical hacking courses that teach their students to be true white-hat hackers rather than just "script kiddies", the skills covered typically include more low-level programming, decompilation and reverse engineering of executable code, analysis of memory and register contents, scripting, and even the capture of malware to discover and dissect new exploit techniques.

Low-level programming skills are essential for anyone wanting to create customized and optimized machine executable code to try to inject into a system. It also can be a useful skill for anyone attempting to disassemble device drivers in order to exploit network cards directly rather than initially attacking the system as a whole. Finally, though difficult and time consuming, low-level programming skills can allow one even to disassemble key components of closed source / proprietary operating systems and applications in order to try to find previously unknown, yet exploitable, security flaws.

Disassembled or even fully decompiled executable programs can provide the hacker with significant information about the libraries and other external programs being depended upon, in addition to the security precautions that are being taken with the data these external pieces of code might be returning. A full reverse engineering of a program's security can be very time consuming, but likely will expose security flaws that were not detected by other more simple means - including extensive penetration testing.

In some cases, though, a full reverse engineering of source code is not required if a hacker can acquire sufficient data memory and register values from during program execution. These values, particularly if their changes can be observed, can provide significant clues into how the program's security works, even if the code itself cannot be examined for whatever reason. Sandbox environments can be used to help obtain these memory and register values for analysis. Further, it might also be presumed that if one is able to observe more than just memory dumps, if one is able actually to read the memory directly during execution on a physical system rather than just in a sandbox, that one also will be able to alter that same memory and thereby compromise that system.

Scripting skills are useful also to a hacker in order to allow fast creation of customized automated attack tools [62]. Indeed, the very source of the term “script kiddies” has its source traced back to these types of attack scripts, written by skilled hackers and then handed off to those who are able to use them to find and exploit security flaws, even if they are themselves incapable of writing such scripts themselves. Scripting skills are useful also for the white-hat hacker to be capable of directly reading some of the various malicious programs that have been created over the years using nothing but scripting, including the well known and notorious ILOVEYOU virus [13].

In many cases, a new attack technique also may be observed when a particular piece of malware is examined that may not have been taught previously in that particular ethical hacking course. It therefore increasingly is common to teach how to capture examples of malware in order to dissect them and discover any new exploit techniques that they might be using. Such practices are paying off, too; the Stuxnet worm that was first captured in July, 2010, is a perfect example, and is now being thought to be the world’s first nation-state created cyber warfare weapon designed to penetrate and sabotage a single specific industrial facility [2]. Some ethical hacking courses may even encourage their students to try to create new attack techniques or even simply recombine older techniques into more effective attacks. These techniques are then tried out on the isolated test environment systems and networks being used by the course and their results are evaluated. Many new and highly effective attack techniques may be developed during such a process, to the dismay of those who already might be uncomfortable with such a course in the first place, but these experiments provide an invaluable leg up on finding defenses against such attacks that likely would be developed someday by true black-hat hackers eventually. The question of whether students should be similarly allowed to examine, modify, and try to improve something as advanced and escape-prone as Stuxnet is a topic that will likely merit very careful consideration.

Section 5: Related Educational Theory

Educational psychology and instructional design theory both advocate a “repetition and feedback” approach to the teaching of critical and complex topics [3, 14, 15]. A student should be guided to practice these critical and complex topics with suitable observation of their progress and prompt feedback on both their successes and their failures. In the instances of success, they should be told they have succeeded at the current level of expectations. After they have suitably demonstrated mastery of the topic, they should be considered to have completed that portion of their education and allowed to move on. In those instances where failure is observed, though, they should be given prompt feedback and further instruction on the topic in question, then again allowed to continue to practice under observation in pursuit of success and progress towards eventual mastery of the topic. Many educational psychologists and instructional design theorists hold that this repetition and feedback cycle, followed until the expected degrees of success are achieved, eventually will lead to mastery of the topic by those willing and able to follow the process through to that end. While heavily used in primary education and secondary education, this theory is just as applicable to a four year undergraduate education.

Educational psychology also holds that if unacceptable performance is allowed to be repeated for any period of time without appropriate feedback and correction, then the student may come to believe that level of performance to be fully acceptable. Any attempt to correct the performance in the future may be met with both confusion and resistance. The confusion likely can be overcome only through more education, during which the student likely will ask ‘well, why didn’t you teach me this correctly the first time!?’ The resistance likely will take repetition to overcome, perhaps even more repetition than it would have taken to teach the acceptable level of performance correctly earlier on. The bad habit of substandard performance will be harder to break after it has become a practiced bad habit and even harder still to break if it has gone on long enough to become second nature to the student. If the habit-modifying repetition does not last long enough to overcome the second nature of the student, then said student most likely will lapse back into the bad habit in the future after they are no longer closely being observed in that facet of their performance.

The interaction of these repetition and feedback theories with code security is in the area of the feedback. If computer science students are allowed repeatedly to construct and submit insecure programs during the formative first few semesters of their computer science education without ever being given feedback on the poor security qualities of the programs, then their likelihood of learning and consistently using more secure coding techniques either then or later is decreased significantly. If, however, the same students are given prompt and accurate feedback about the security qualities of their code and corrected, and then given further instruction in the case of any failures, they are far more likely both to achieve mastery of code security and consistently produce secure code for industry throughout their whole career.

Secure code is important but secure software designs are just as important. In computer science education, coding skills normally are used as part of the basis for teaching software design. If secure coding is taught at the beginning of the computer science education, then subsequent efforts to teach the student the importance of and techniques for secure software design, most likely in a Software Engineering course, should be much more successful. This also is another application of the repetition and feedback theory. The repetition in question is now a more long term form of repetition, but it still is valid repetition of security theory, even if a slightly different area of study within the overall topic of computer security theory. It also is repetition of the importance of security and it is at the same time providing a morale-boosting form of feedback for the student that they successfully have mastered a previous piece of security theory. Even though secure software design theory will have its own internal repletion and feedback cycles, it serves as the culminating cycle for the majority of computer science majors that will not specialize in security, while it serves as a second cycle for those who will go on to advanced studies in information security, network security, and / or system security - which will, of course, also of course have their own repetition and feedback cycles.

Either through its emphasis or its de-emphasis, all computer science education has repetition and feedback regarding computer security. If the next generations of computer science students are to be any better at producing secure software than their predecessors who currently are producing buggy, insecure software out in industry, then security must receive more repeated emphasis and feedback in computer science education.

Further, there are some in the current generation of computer science students that will someday go on to teach computing to future students; for them to have a better understanding of security will help perpetuate and improved understanding of security. This in turn means that security must be required of all computer science students as soon as that is possible.

Section 6: Related Business Theory

Total Quality Management theory, from W. Edwards Deming corresponds with the issue of code quality, as well [41]. Deming theorized that product quality could be increased and errors in production could be reduced by keeping in conformance to well designed requirements and through high quality worker training. The Total Quality Management process for improvement also involves a detailed examination of the entire production system, from start to finish, in order to locate areas of inefficiency and potential causes for error. Root Cause Analysis, which calls for the root or most basic cause for a problem to be sought after, is another similar and related business theory [41].

The application of Total Quality Management theory and Root Cause Analysis theory to the current state of computer security identifies computer software, at best, as having inconsistencies in security quality. These quality issues appear as though they are due to errors in design and coding. Thus it follows that the quality of the training of software developers, the requirements they are meant to follow, and the conformance with those requirements need to be examined in order to find the most likely root causes for the quality issues. In this particular situation, though, the software developers themselves are the ones who generate the detailed requirements they follow and control how closely they conform to said requirements. Thus it would seem that the quality issues lay firmly in the training of - and consequently the performance of - the software developers. Therefore, the logical conclusion is that it is the training of the software developers which should be improved in order to fix the software security quality issues that industry currently faces.

If the software developers can be trained better in regards to the importance of security, then said developers likely are to pay more attention to it in their work. If the software developers can be trained better in regards to the need to design programs so that they are secure from the start - rather than trying to patch in security later - then they are more likely to do so. If the software developers can be trained better in regards of which coding constructs to use and not to use, then they are more likely to write higher quality code. If the software developers can be trained better in regards to the proper use of tools to aid in the creation of more secure programs, then they are more likely to use those tools. If the software developers can be trained better in regards to the need for arguing that "Total Quality Management" demands they "invest the time and resources" to "insure" the program is secure, then their managers are going to understand they are talking about sound business principles (TQM, investment, insurance), rather than just spouting 'technobabble' to pad their budget.

In the end, it is management that must decide whether computer security problems need to be solved and - if needed - decide how they need to be solved. If management comes to the conclusion that the academic preparation of their software developers is the root cause, then those managers are more likely to try to hire more developers who are highly competent in computer security, particularly code security, in the their future hires. The universities that are requiring security, particularly those universities requiring code security, for all of their computer science undergraduate students naturally will develop a competitive advantage over those universities that do not require as much security for their computer science undergraduate students. Considering that some surveys suggest that the supply of domestic students interested in pursuing an undergraduate computer science major may continue to decline in the near future, any legitimate competitive advantage that a particular university's computer science department can develop over other universities' computer science departments should be considered desirable.

CHAPTER III

METHODOLOGY

The author taught an introductory course in computer security to computer science and computer information science majors in the Department of Computer and Information Sciences at Southwest Baptist University in the spring 2010 semester. The course was approved by Southwest Baptist University to be taught as CIS-2953 (Special Topics: Secure Software). The course syllabus can be found in Appendix D. There were 23 students enrolled in the course at the beginning of the semester, 20 of whom successfully completed the course. The purpose of the course was fivefold.

The first course goal was to expose the students to the importance of security. This was accomplished by a series of case studies of historical security failures. This goal should be a portion of any serious introduction to security and is not unique from contemporary computer science courses in security. Therefore, it was not assessed.

The second course goal was to expose the students to the importance of security as early in their degree program as possible. This would leave them the maximum possible time remaining in their undergraduate program to watch for security implications and use their security skills in other courses. The course's only prerequisite was Computer Science I. This enabled some freshman level students to take the course. Unfortunately, due to the time constraints the author was operating under, this second goal was not assessed as a portion of the research project.

The third course goal was to familiarize the students with common categories of security-related programming flaws. Industry has identified and categorized common security-related programming flaws into several different groupings. The author drew primarily from [34], and covered 14 categories that a student coming directly from Computer Science I should be able to understand. This goal was the primary focus of the assessment.

The fourth course goal was to teach the students how to write secure code. This goal dealt more with the software development process than the programming flaws of the previous goal. The author drew primarily from [49]. This goal also has been assessed within this dissertation.

The fifth course goal was to teach the students how to write secure code before they became accustomed to writing less secure code. Unfortunately, again due to the time constraints the author was operating under, this goal was not assessed as a portion of this research project.

The course was taught using the Java programming language. Java is not the ideal choice for a course dealing with security-related programming flaws because of how Java handles memory, pointers, strings, and exceptions. Either C or C++ would have been a preferred choice of language, due to the significantly larger number of creative mistakes they allow with memory, pointers, and strings. Despite this, the author was constrained to use Java in this research project because of the choice only to require the completion of Computer Science I before attempting the course. (Southwest Baptist University teaches Computer Science I in Java, not unlike Oklahoma State University.) In the future, the author would like to experiment with teaching a course that combines an introduction to security with an introduction to C language, but that would have to be a different project; unfortunately, the teaching of the C language would complicate the assessment of the course and is beyond the scope of this project.

The course generally ran as the author had expected it to run. There were minor revisions to the expected course schedule, which can be found in Appendix E. These schedule changes mainly were due to the inclusion of in-class code reviews of programs submitted by students in the course. The author removed any personally-identifable information from the submitted programs prior to the code reviews and allowed students to opt-out of having their code publicly, though annonomously, reviewed in class without any repercussions. None of the students requested to opt-out and these in-class code reviews were some of the most active class discussions throughout the entire semester. The first three programs were written completely by the students. The last three programs were based on code provided to the students by the author. These pieces of starter code for programs four though six can be found in Appendix F through H respectively.

Assessment was collected in the form of a pre-test survey and an identical post-test survey given to both the participating students and control populations not enrolled in the course. The survey instrument can be found in Appendix C. The data obtained from the analysis of the surveys can be found in Appendix I. The pre-test was administered the first week of the semester, and the post-test was administered the last week - prior to finals. Participation in these in-class tests was voluntary for both the students enrolled in the CIS-2953 course and the control populations not enrolled in the CIS-2953 course. The control populations were CIS-1154 (Computer Science II), CIS-2213 (Introduction to System Analysis and Design), CIS-3323 (Database Management Systems Design), and CIS-4443 (Networks). These four courses were selected to serve as control populations because they contain the vast majority of Southwest Baptist University's computer science and computer information science majors and minors; they also were chosen to minimize the number of impacted courses. There was not a cumulative review in the CIS-2953 course before the post-test. In addition, the students were not specifically instructed to review certain topics in preparation for the post-test, as they would have been if it were a test upon which their grade would be partially dependent.

Each pre-test was assigned a unique six digit pseudo-random number to insure anonymity. Participants were instructed to record their assigned six digit pseudo-random number in a file within their personal, private network drive space for reuse during the post-test. The Southwest Baptist University Information and Technology Services department performed sufficient backup maintenance of these network drives to insure against loss of the pertinent data between the pre-test and post-test. This storage was deemed to be more secure than having the students recorded the numbers in notebooks or textbooks, where the author might accidently see them at some point during the semester. To further insure anonymity, these numbers do not appear anywhere within this dissertation.

The pre-tests and post-tests were required to be administered by individuals who were not a "supervisor, teacher, or employer" in regards to any of the participating students. This excluded the entire faculty and staff of the author's department and college, among others. The individuals who were chosen to perform the task of administering the surveys were the Southwest Baptist University Director of Institutional Effectiveness and the Southwest Baptist University Assessment Coordinator, two individuals who, alongside other duties, are responsible for overseeing faculty evaluations. The test answers were then typed by Southwest Baptist University staff so that the author would not be able to identify participants through recognition of their handwriting, and the original answer sheets were never shown to the author. The participants were not asked to sign proof of informed consent in order to insure that the author would not be able to determine who volunteered to participate and who declined to participate. An informational script was read to the subjects, prior to the surveys, to inform them of their rights. This script may be found in Appendix A. An informational handout also was provided for the subjects to take with them in case they should desire further information about their rights than was provided in the informational script. This handout may be found in Appendix B. These elaborate measures were required by and approved by the Southwest Baptist University Research Review Board and the Oklahoma State University Institutional Review Board.

CHAPTER IV

FINDINGS

Section 1: Anticipated Findings

The author expected the project’s primary assessment goal, involving the understanding of security, to reveal five specific findings. The author expected the project’s secondary assessment goal, involving coding skills related to security, to reveal five very similar findings to the findings of the project’s primary assessment goal.

First, the author expected to find no statistically significant differences between the performance of the enrolled test subjects and the control subjects on the pre-test. Any differences in performance between these two populations on the pre-test were projected to be due primarily to the self selection of the two populations, based on their interests in security.

Second, the author expected to find no statistically significant differences between the performance of the control subjects on the pre-test and the control subjects on the post-test. Any differences in performance between these two tests were projected primarily to be due to information leakage from the test subjects, based on the relatively tight-knit nature of the department's small population, and secondly due to any security topics that might be incidentally covered in other computer science courses, which the control subjects might be enrolled in during the time period of the study.

Third, the author expected to find statistically significant differences between the performance of the test subjects on the pre-test and the test subjects on the post-test. Any differences in performance between these two tests were projected to be due primarily to the CIS-2953 (Special Topics: Secure Software) course that the test subjects were enrolled in during the time period of the study, and secondly due to any security topics that might incidentally be covered in other computer science courses the test subjects might be enrolled in during the time period of the study.

Fourth, the author expected to find statistically significant differences between the performance of the test subjects and the control subjects on the post-test. Any differences in performance between these two populations on the post-test were projected to be due primarily to the CIS-2953 (Special Topics: Secure Software) that the test subjects were enrolled in during the time period of the study and secondly due to the self selection of the two populations based on their interests in security.

Fifth, the author expected to find statistically significant differences between the improvement of performance of the test subjects and the control subjects on the post-test compared to the pre-test. Any differences in improvement of performance between these two populations on the post-test over the pre-test were projected to be due primarily to the CIS-2953 (Special Topics: Secure Software) the test subjects were enrolled in during the time period of the study and secondly due to the self selection of the two populations based on their interests in security.

In addition, the author expected to observe differences in the percentage of correct responses to vary from answer to answer, and believed that analysis of such might provide additional findings. The author expected that due to low population sizes, the significance of any results might not be reliable, but was unable to mitigate this potential problem while running the study at the chosen location.

Section 2: Actual Findings

As shown in figure 4.1, the test subjects performed better than the control subjects on the multiple choice section of the pre-test measuring the primary assessment goal, involving the understanding of security, but the difference was not statistically significant. This was in line with the anticipated finding in this area. This supports the author’s theory that despite the two populations being self-selected, any differences between them were not statistically significant.

As shown in figure 4.1, the test subjects performed worse than the control subjects on the coding section of the pre-test which measured the secondary assessment goal, involving coding skills related to security, but the difference was not statistically significant. This was in line with the anticipated finding in this area. This supports the author's theory that despite the two populations being self-selected, that any differences between them were not statistically significant.

	Minimum Score	**Maximum Score**	**Average Score**	**Standard Deviation**
Test Subjects				
Multiple Choice Only	7%	47%	28.3%	10.7%
Code Problems Only	0%	33%	12.7%	16.6%
All Problems	6%	39%	25.7%	10.0%
Control Subjects				
Multiple Choice Only	13%	40%	24.7%	8.0%
Code Problems Only	0%	33%	19.4%	16.8%
All Problems	11%	39%	23.8%	8.1%

	Percent Difference Of Average Scores	**2 Tailed P-Value**
Test Subjects vs Control Subjects		
Multiple Choice Only	13%	0.21
Code Problems Only	42%	0.18
All Problems	7%	0.50

Table 4.1: Pre-Test Results (Test Subjects vs Control Subjects) with percent difference as the absolute value of the difference of the averages over the average of the averages [55]

As shown in figure 4.1, when the overall performance of the test subjects was compared to the overall performance of the control subjects in regards to all problems within the pre-test, the percent difference in performance between the two populations was less than it was for both the multiple choice section and the coding section. In fact, the difference in performance was less statistically significant than that of either the multiple choice section or the coding section. This supports the author's theory that, despite their being self-selected, that any differences between the two populations were not statistically significant.

	Minimum Score	Maximum Score	Average Score	Standard Deviation
Pre-Test				
Multiple Choice Only	13%	40%	24.7%	8.0%
Code Problems Only	0%	33%	19.4%	16.8%
All Problems	11%	39%	23.8%	8.1%
Post-Test				
Multiple Choice Only	7%	33%	20.3%	7.2%
Code Problems Only	0%	33%	12.5%	16.5%
All Problems	6%	33%	19.0%	7.5%

	Percent Difference Of Average Scores	2 Tailed P-Value
Post-Test vs Pre-Test		
Multiple Choice Only	18%	0.03
Code Problems Only	36%	0.13
All Problems	20%	0.01

Table 4.2: Control Subject Results (Post-Test vs Pre-Test) with percent difference as the absolute value of the difference of the averages over the pre-test average [55]

As shown in figure 4.2, the control subjects performed worse on the multiple choice section of the post-test than on the multiple choice section of the pre-test, which measured the primary assessment goal that involved the understanding of security. This difference in performance was statistically significant. The author attributes the decline in performance on the post-test primarily to the times of the semester in which the two tests were administered. The pre-test was administered during the first week of the semester, which typically is a low stress period for most students. Conversely, the post-test was administered during the last week before finials, which typically is an extremely stressful time for most students. The author does not know how this risk could have been mitigated during the single semester duration of this study; however, were a multi-semester version of this study to be run, the author believes this risk of error could be mitigated by administering the surveys at the same time during each of the semesters.

As shown in figure 4.2, the control subjects did perform worse on the coding section of the post-test than on the coding section of the pre-test, which measured the secondary assessment goal, involving coding skills related to security; despite this, the worse performance was not statistically significant. The author suspects the decline in performance on the post-test to be due primarily to the times of the semester in which the two tests were administered but notes it may be instead simply due to error.

As shown in figure 4.2, when the overall performance of the control subjects on all problems of the post-test was compared to the overall performance of the control subjects on all problems of the pre-test, the change in performance was more statistically significant. This suggests the author's belief that the performance decline was not due to error, and more realistically due to either the times of the semester in which the two tests were administered or some other undetected influence.

As shown in figure 4.3, the test subjects performed better on the multiple choice section of the post-test than on the multiple choice section of the pre-test, which measured the primary assessment goal, involving the understanding of security. This measure of performance was statistically significant. This improvement was in line with the anticipated finding in this area; it also suggests that this course goal was indeed achieved to some degree, even if the magnitude of the change was not as large as the author may have hoped.

As shown in figure 4.3, the test subjects performed almost exactly the same on the coding section of the post-test as on the coding section of the pre-test, which measured the secondary assessment goal, involving coding skills related to security. This performance was not statistically significant. The author believes that the lack of a statistically significant improvement by the test subjects clearly indicates this course goal was not achieved.

	Minimum Score	**Maximum Score**	**Average Score**	**Standard Deviation**
Pre-Test				
Multiple Choice Only	7%	47%	28.3%	10.7%
Code Problems Only	0%	33%	12.7%	16.6%
All Problems	6%	39%	25.7%	10.0%
Post-Test				
Multiple Choice Only	13%	60%	34.0%	13.5%
Code Problems Only	0%	67%	12.7%	19.7%
All Problems	11%	61%	30.4%	13.8%

	Percent Difference Of Average Scores	**2 Tailed P-Value**
Post-Test vs Pre-Test		
Multiple Choice Only	20%	0.05
Code Problems Only	0%	1.00
All Problems	19%	0.11

Table 4.3: Test Subject Results (Post-Test vs Pre-Test) with percent difference as the absolute value of the difference of the averages over the pre-test average [55]

As shown in figure 4.3, when the overall performance of the test subjects on all problems of the post-test was compared to their overall performance on all problems of the pre-test, the results on the two different parts canceled each other out, producing a difference that was both smaller in magnitude and not statistically significant. The author suspects this view of the results to be less important than the two subcomponents that produce it.

As shown in figure 4.4, the test subjects performed better than the control subjects on the multiple choice section of the post-test, which measured the primary assessment goal, involving the understanding of security. This difference in performance was statistically significant. This was in line with the anticipated finding in this area. The improvement also suggests that this course goal, indeed, was achieved.

	Minimum Score	**Maximum Score**	**Average Score**	**Standard Deviation**
Test Subjects				
Multiple Choice Only	13%	60%	34.0%	13.5%
Code Problems Only	0%	67%	12.7%	19.7%
All Problems	11%	61%	30.4%	13.8%
Control Subjects				
Multiple Choice Only	7%	33%	20.3%	7.2%
Code Problems Only	0%	33%	12.5%	16.5%
All Problems	6%	33%	19.0%	7.5%

	Percent Difference Of Average Scores	**2 Tailed P-Value**
Test Subjects vs Control Subjects		
Multiple Choice Only	50%	0.00
Code Problems Only	2%	0.97
All Problems	46%	0.00

Table 4.4: Post-Test Results (Test Subjects vs Control Subjects) with percent difference as the absolute value of the difference of the averages over the average of the averages [55]

As shown in figure 4.4, the test subjects performed better than the control subjects on the coding section of the post-test, which measured the secondary assessment goal, involving coding skills related to security; however, the improvement was not statistically significant. The author suspects the lack of a significantly improved performance by the test subjects indicates this course goal was not achieved.

As shown in figure 4.4, when the overall performance of the test subjects on all problems of the post-test was compared to the overall performance of the control subjects on all problems of the post-test, a statistically significant difference still was observed.

	Minimum Score	Maximum Score	Average Score	Standard Deviation
Test Subjects				
Multiple Choice Only	7%	13%	5.7%	12.5%
Code Problems Only	0%	33%	0.0%	18.3%
All Problems	6%	22%	4.8%	12.9%
Control Subjects				
Multiple Choice Only	-7%	-7%	-4.4%	9.2%
Code Problems Only	0%	0%	-6.9%	21.7%
All Problems	-6%	-6%	-4.9%	8.9%

	Percent Difference Of Average Scores	2 Tailed P-Value
Test Subjects vs Control Subjects		
Multiple Choice Only	1600%	0.00
Code Problems Only	200%	0.35
All Problems	19400%	0.01

Table 4.5: Improvement of Results from Pre-Test to Post-Test (Test Subjects vs Control Subjects) with percent difference as the absolute value of the difference of the averages over the average of the averages [55]

As shown in figure 4.5, the test subjects performed better on the multiple choice sections of the post-test versus the pre-test, while the control subjects performed worse. This result was statistically significant. While this first appears to be in line with the anticipated finding in this area, the author considers this result tainted by the decline in performance of the control subjects, and believes this finding to be unreliable.

As shown in figure 4.5, the test subjects performed almost exactly the same on the coding sections of the post-test over the pre-test, while the control subjects performed worse. In addition, the differences were not statistically significant. The author believes this finding to be unreliable.

As shown in figure 4.5, when the overall performance improvement of the test subjects on all problems of the post-test was compared to the overall performance improvement of the control subjects on all problems of the post-test, a statistically significant difference still was observed. While this first appears to be in line with the anticipated finding in this area, the author considers this result tainted by the decline in performance of the control subjects, and believes this finding to be unreliable.

In addition to verifying the anticipated findings related to the primary and secondary assessment goals, a detailed analysis of the frequency of correct response to certain individual answers produced some supplementary, yet notable, findings.

While comparing the results of the pre-test and post test, the largest improvement observed within the test subjects was on answer 17D where eleven additional students noted the existence of a potential race condition in the sample code. The improvement also suggests that this component of primary assessment goal was indeed achieved to some degree.

Additionally, there was a noticeable improvement with the test subjects observed between the pre-test and post-test on question 7, where several additional test subjects noted that code-defect risk analysis should begin before coding process begins. The improvement also suggests that this component of primary assessment goal was indeed achieved to some degree.

Finally, there was also a noticeable improvement with the test subjects observed between the pre-test and post-test on answer 4D, where seven additional students noted that the java.util.Random class should never be considered cryptographically strong. The improvement also suggests that this component of primary assessment goal was indeed achieved to some degree.

CHAPTER V

FUTURE WORK

Section 1: Future Work in Code Security

As the author prepared to teach this course, certain problems arose; unfortunately, the author could not address these problems without overly complicating the research. The first issue involved the programming language with which the course was to be taught. Java hardly is an ideal language for the course, yet it is the language the author was forced to use based on the course's prerequisite. As Dewar and Astrachan put it:

> "It's not impossible to teach the fundamental principles using Java, but it's a difficult task. The trouble with Java is twofold. First it hides far too much, and there is far too much magic. Students using fancy visual integrated development environments working with Java end up with no idea of the fundamental structures that underlie what they are doing. Second, the gigantic libraries of Java are a seductive distraction at this level. You can indeed put together impressive fun programs just by stringing together library calls, but this is an exercise with dubious educational value." [22]

C or C++ would be far better choices for teaching security, as Seacord and as Mano, et al., have noted:

> "Although the flexibility and performance of C and C++ aren't in question, security has increasingly become an issue." [59]

> "The buffer overflow problem is easily taught using C/C++, but not so in newer languages such as Java and C#, which perform bounds checking on inputs. In fact, these languages are designed to prevent buffer overflow vulnerabilities. This brings up two questions. Why should we teach the buffer overflow problem if newer languages are not susceptible? Furthermore, how should we teach it if the class uses a language such as Java?" [44]

The author always has wanted to introduce Southwest Baptist University students to C and C++ earlier than the point at which it traditionally is taught, their junior year. After all, the author taught CS-2432 using C to students at Oklahoma State University who had never programmed before and now Oklahoma State University is teaching CS-2433 in C and C++ to the same level of students. Why could the author not teach a secure programming course in C and C++ to students who already knew how to program in Java from Computer Science I?

Lastly, the author believes the best assessment of an elective secure programming course similar to the one used in this study - even when modified to use C and C++ as noted above - would be though a department-wide pre-test survey at the beginning of the students' second semester in the department, followed up by a post-test survey at the beginning of the students' last semester in the department. A more effective mechanism for tracking which students had participated in the course and which students had not would have to be devised to pass the approval of the relevant human subject research oversight boards; this is due to the fact that multiple students did not retain their 6 digit pseudo-random number, even just for the duration of just a single semester class, and therefore had to be excluded from this study.

Section 2: Future Work in Computer Science Curriculum

Many possible solutions for the problem of computer security have been investigated over the years, and many more are likely to be investigated in the future. The author does not necessarily propose any truly *new* curriculum ideas, but rather a *hybrid* of some elements contrived from current curriculum theories and a code security course inspired by industry's buggy code theory. At the same time, the author clearly rejects certain other elements of and generally ignores other elements of current curriculum theories as they are irrelevant to the author's curriculum proposals.

The author believes that elective information security, network security, and system security (including ethical hacking material) coursework really should be available at those universities which offer graduate degrees in computer science. The author also believes that all master's degree students should be strongly encouraged, if not outright required, to take a graduate level security course that includes elements of information security, network security, system security, ethics, and the law. The author believes that all doctoral students should be required to take such a graduate level security course as well. The author also believes, however, that these graduate level computer science curriculum issues are generally ignorable in the context of the author's real curriculum proposals which focus on the undergraduate computer science curriculum.

Security must become a required part of the computer science undergraduate curriculum. It is true that some elements of security should be integrated throughout the curriculum, but only as they naturally fit. Security should not be forced into other courses simply to eliminate the need for removing any courses from the required curriculum in order to make room for a required security course. Some courses, perhaps such as numerical analysis, may not have any natural security tie-ins and consequently may never introduce any security. On the other hand, software engineering, operating systems, file systems, networks, and database courses - among others - have natural tie-ins and therefore should have some related security integrated into their content. Many textbooks can be found that already explicitly provide for such integration.

Security should be introduced as early in the curriculum as possible. Security for non-majors should be integrated into whatever course functions as each university's Introduction to Computer Science course, be that a Computer Science 0 overview course or be that a more traditional Computer Science I course. Computer Science I should provide a solid foundation in design and coding for future coursework in secure programming, but should not overtly stress a security focus for its design and coding topics. Rather, Computer Science I should approach design and coding from the perspectives of what is achievable in theory, what is functional in practice, and what is robust under general adverse conditions - rather than under attack conditions. Computer Science I should be taught from the perspective of promoting robust code. Later security courses can point out that more robust code equals more secure code and that less robust code equals less secure code. After Computer Science I the student, in the continuation of a previous analogy, should be ready to have their training wheels taken off of their programming skills bicycle and start moving on to learning how to do more advanced things such as pull a wheelie … or learn how to write a priority queue in Computer Science II and learn about secure programming in a code security course.

An introduction to secure software development should be taught directly following the student's completion of Computer Science I. To do so immediately after Computer Science I will minimize the time they have to develop and practice bad habits. The secure programming course should include more advanced general computer security theory than their previous computer security course for non-majors covered. It also would discuss secure design and coding but would stress the coding more than the design, under the assumption that secure design should be further reinforced in the student's future software engineering coursework. The study of secure coding should begin in the language the student used in Computer Science I and then move on to include other similar languages as well. Specifically, the student needs to be taught what common pitfalls exist in very simplistic and fundamental languages such as C language, due to the creative problems that comparable languages allow that other, more modern languages such as Java do not allow. The student should be given sufficient programming assignments to provide educationally sound repetition and feedback on their secure programming skills.

A student should be taught the importance of security. Marks and Stinson suggest that security may be one of the most important aspects of computer science:

> "In today's development environment, security is more important than execution speed, user interface, or even time needed to write the code." [45]

As long as academia continues to graduate students that have not been taught the importance of security, then security problems will continue to arise in an unnecessarily high rate in industry. While security problems will likely never be completely eliminated, their frequency and the magnitude of their impact can be reduced. The impact of increased security upon execution speed can largely be mitigated by the rapid expansion of the number of CPU cores and pipelines available to the standard personal computer. The demands of improved security upon user interfaces may require more careful limits on what can be displayed based upon a user's permissions, but the techniques necessary to enforce such limitations through code are well known and easily implementable if enough time and careful attention are taken in a program's design and coding stages. Additionally, students need to understand that real programs cannot be designed and written "overnight" (either literally or figuratively). Security must receive its just due, even if that means convincing management to push back the release date from what had been previously advertised. After all, it is far better to inform the public that the company took extra time to make sure that the program's security was done right, rather than to release buggy code that causes the company's customers major problems in the future.

The student should be taught how to use available tools that can help produce more secure code, whether these tools were designed to be security related tools or not. The allegedly neigh-impossible needs to be differentiated from the code-enforced truly impossible and the neigh-impossible should actually be taught to be routinely expected given sufficient real world usage. Lastly, Bishop and Frinke suggest the student needs to learn not to trust blindly, but rather to assume that every user and every program is out to exploit their code.

> "It has been said that in computer security, paranoia is not enough, but it is a good place to start." [7]

CHAPTER VI

CONCLUSION

At present, computer security problems are out of control. Something must be done to reduce the frequency and magnitude of the security problems that the world is experiencing. While a complete and permanent solution is desirable, the problems are severe enough to make even partial and temporary solutions worth trying.

Security-aware compilers and post-processors can help, but only if developers will use them properly. Unfortunately, developers have spotty track records for using lint and other similar general purpose compilation aids. This leads to doubt about the probable usage rates of security-aware compilers and post-processors; which in turn leads to doubt in their potential effectiveness as a complete solution to the security problem. Despite this, it is a valid partial solution, and therefore definitely worth pursuing.

Education is another area in which significant potential for improvement may be found. Industry is pursuing considerable continuing education efforts to correct deficiencies in their developers' academic preparations. While these educational initiatives in industry are clearly valid partial solutions, they would have never been so necessary if the developers had received the proper education in computer security from academia.

Computer security, particularly code security, should be required as early as possible, preferably during the first year of the undergraduate computer science curriculum. All computer science majors must be competent coders, capable of and well practiced in writing secure code, before they graduate. In addition, all computer science majors must be aware of computer security's importance, particularly that of code security.

Security can be, and should be, integrated into certain other computer science courses, but only as it naturally relates to the primary focus of said courses. Also, security should have its own courses that would reinforce the importance of security as an area within computer science; this also would allow students at universities that have multiple security courses to select security as their area of emphasis. At the very least, every computer science curriculum should have and require a code security course. Additionally, those universities that allow a student to focus on security also should have elective information security, network security, system security courses, or a combination thereof.

The author has taught an introductory security course with emphasis in code security over the course of one semester. The students in the course ranged from second semester freshman, straight out of Computer Science I, to seniors graduating at the end of that semester. While results from the pre-test and post-test surveys completed by course subjects were mixed, they suggested that the course was at least partially successful. The students did seem to have a better understanding of computer security but seem to have not improved as much within the area of secure coding as the author had anticipated. The author feels that more repetition and feedback on the writing of secure code will improve the course the next time it is offered.

The author also believes that a pre-test survey of a larger population of second semester freshman, straight out of Computer Science I, and a post-test of those same students immediately prior to graduation would show a more accurate assessment of the impact of either taking such a course or not taking such a course. To make such an assessment, though, would require a larger student population than the author has close access to. It also would involve approximately half of the subjects not taking such a course, in order that they may act as a valid control group, and the author would not think it best for any computer science student to skip taking such a course, particularly just for the sake of researching the course's impact. The author considers the computer security problem as better addressed with an imperfect solution applied to more students than a better solution applied to fewer students, even if it only is for a few years of research.

The author believes these proposals are not a perfect solution for the present computer security problem. However, the author does believe that these proposals are a valid partial solution. While the author believes these proposals should be immediately and universally implemented, the author recognizes the fact that this schedule is not a realistic possibility, and expects that additional research project implementations of these proposals will create opportunities for their further refinement.

REFERENCES

[1] Adams, E., Barr, V., Fendrich, J., Leska, C., Thomas, B., Towhidnejad, M., Teaching software testing throughout the curriculum, Proceedings of the 5th annual CCSC Northeastern Conference, 178-180, 2000.

[2] Adhikari, R., Technology news: suspicions rise: has a cyberwar started?, 2010, www.technewsworld.com/story/Stuxnet-Suspicions-Rise-Has-a-Cyberwar-Started-70892.html?wlc=1285453582, retrieved September 24, 2010.

[3] Astin, A., Student involvement a developmental theory for higher education, 1984, www.middlesex.mass.edu/TutoringServices/AstinInvolvement.pdf, retrieved July 5, 2010.

[4] Ayoub, R., Chin, A., Parberry, I., A new model for a student cyber security organization, Proceeding of the 2nd annual conference on information security curriculum development, 12-15, 2005.

[5] Bailey, M., Injecting programming language concepts throughout the curriculum, ACM SIGPLAN Notices, 45, (11), 36-38, 2008.

[6] Bailey, M., Coleman, C., Davidson, J., Defense against the dark arts, Proceedings of the 39th SIGCSE technical symposium on Computer science education, 315-319, 2008.

[7] Bishop, M., Frincke, D., Teaching robust programming, IEEE Security & Privacy, 3, (2), 54-57, 2004.

[8] Bishop, M., Frincke, D., Teaching secure programming, IEEE Security & Privacy, 54-56, 2005.

[9] Bogolea, B., Wijekumar, K., Information security curriculum creation: a case study, Proceedings of the 1st annual conference on Information security curriculum development, 59-65, 2004.

[10] Bonakdarian, E., White, L., Robocode throughout the curriculum, Journal of Computing Sciences in Colleges, 21, (3), 311-313, 2004.

[11] Brechner, E., Things they would not teach me of in college: what Microsoft developers lean later, Companion of the 18th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, 134-136, 2003.

[12] Carlson, D., Teaching computer security, ACM SIGCSE Bulletin, 38, (2), 64-67, 2004.

[13] CERT, CERT advisory CA-2000-04 love letter worm, 2000, www.cert.org/advisories/CA-2000-04.html, retrieved September 24, 2010.

[14] Chickering, A., Ehrmann, S., Implementing the seven principles: technology as lever, 2004, www.fmtsystems.com/04-news/Impl-7-prin.pdf, retrieved July 5, 2010.

[15] Chickering, A., Gamson, Z., Seven principles for good practice in undergraduate education, 1987, crunchie.tedi.uq.edu.au/blendedlearning/pdfs/fall1987.pdf, retrieved July 5, 2010.

[16] *CIS 1103 - Intro to Computing: Southwest Baptist University 2009 - 2010,* New York: NY: Pearson Custom Printing, 2009.

[17] Cook, J., What C.S. graduates don't learn about security concepts and ethical standards or - "Every company has its share of damn fools. Now every damn fool has access to a computer", Proceedings of the 17th SIGCSE technical symposium on Computer science education, 89-96, 1986.

[18] Cross, T., Academic freedom and the hacker ethic, Communications of the ACM, 51, (6), 37-40, 2006.

[19] Daswani, N., Kern, C., Kesavan, A., *Foundations of Security: What Every Programmer Needs to Know*, Berkeley, CA: Apress, 2007.

[20] Davern, P., Scott, M., Steganography: its history and its application to computer based data files, citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.8482&rep=rep1&type=pdf, retrieved July 9, 2010.

[21] DeClue, T., Computer science 1 - is this your father's Oldsmobile?, The Journal of Computing in Small Colleges, 17, (4), 12-17, 2000.

[22] Dewar, R., Astrachan, O., CS education in the U.S.: heading in the wrong direction?, Communications of the ACM, 54, (7), 41-45, 2009.

[23] Futcher, L., von Solms, R., Guidelines for secure software development, ACM International Conference Proceedings Series, 338, 56-65, 2008.

[24] Gallagher, T., Jeffries, B., Langauer, L., *Hunting Security Bugs*, Redmond, WA: Microsoft Press, 2006.

[25] Grissom, S., iPhone application development across the curriculum, Journal of Computing Sciences in Colleges, 26, (1), 40-46, 2008.

[26] Harrison, W., Hanebutte, N., Alves-Foss, J., Programming education in the era of the Internet: a paradigm shift, Proceedings of the 39th Hawaii International Conference on System Sciences, 219b-219b, 2006.

[27] Herath, A., Herath, S., Goonatilake, R., Herath, S., Herath, J., Designing computer forensics courses using case studies to enhance computer security curricula, Journal of Computing Sciences in Colleges, 25, (1), 2007.

[28] Hjelmås, E., Wolthusen, S., Full-spectrum information security education: integrating B.Sc., M.Sc., and Ph.D. programs, Proceedings of the 3rd annual conference on Information security curriculum development, 5-12, 2006.

[29] Hoglund, G., Security band-aids: more cost-effective than “secure” coding, IEEE Software, 21, (6), 56 & 58, 2002.

[30] Holliday, M., Kreahling, Information security and computer systems: an integrated approach, Proceedings of the 3rd annual conference on Information security curriculum development, 58-63, 2006.

[31] Horton, D., HyperCard throughout the curriculum, Journal of Computing Sciences in Colleges, 7, (5), 49, 1991.

[32] Howard, M., LeBlanc, D., *Writing Secure Code 2nd Ed.*, Redmond, WA: Microsoft Press, 2003.

[33] Howard, M., LeBlanc, D., Viega, J., *19 Deadly Sins of Software Security,* New York, NY: McGraw-Hill/Osborne, 2005.

[34] Howard, M., LeBlanc, D., Viega, J., *24 Deadly Sins of Software Security*, New York, NY: McGraw-Hill, 2010.

[35] Ingham, K., Implementing a successful secure coding continuing education curriculum for industry: challenges and successful strategies, Proceedings of the 19th Conference on Software Engineering Education and Training Workshops, 25-25, 2006.

[36] Karam, O., Peltsverger, S., Teaching with security in mind, Proceedings of the 47th Annual Southeast Regional Conference, Article No. 68, 2009.

[37] Kumar, R., Pandey, S., Ahson, S., Security in coding phase of SDLC, 3rd International Conference on Wireless Communication and Sensor Networks, 118-120, 2007.

[38] Kurose, J., Ross, K., *Computer Networking: A Top-Down Approach 5th Ed.*, New York, NY: Addison-Wesley, 2008.

[39] Lester, C., Jamerson, F., Incorporating software security into an undergraduate software engineering course, Proceedings of the 2009 Third International Conference on Emerging Security Information, Systems and Technologies, 161-166, 2009.

[40] Lester, C., Narang, H., Chen, C., Infusing information assurance into an undergraduate CS curriculum, Proceedings of the 2008 Second International Conference on Emerging Security Information, Systems and Technologies, 300-305, 2008.

[41] Levis, M., Helfert, M., Brady, M., Information quality management: review of an evolving research area, 2010, mitiq.mit.edu/ICIQ/PDF/INFORMATION%20QUALITY%20MANAGEMENT%20--%20REVIEW%20OF%20AN%20EVOLVING%20RESEARCH%20AREA.pdf, retrieved July 8, 2010.

[42] Mader, A., Srinivasan, S., Curriculum development related to information security policies and procedures, Proceedings of the 2nd annual conference on Information security curriculum development, 49-53, 2005.

[43] Mancoridis, S., Software analysis for security, Frontiers of Software Maintenance, Sept. 28 2008-Oct. 4 2008, 109-118, 2008.

[44] Mano, C., DuHadway, L., Striegel, A., A case for instilling security as a core programming skill, 36th ASEE/IEEE Frontiers in Education Conference, 13-18, 2006.

[45] Marks, D., Stinson, M., Security trumps efficiency: putting it into the curriculum, Journal of Computing Sciences in Colleges, 24, (4), 162-169, 2007.

[46] McAfee KnowledgeBase, McAfee DAT 5958 False Positive Error, kc.mcafee.com/corporate/index?page=content&id=KB68787, retrieved July 9, 2010.

[47] McCullagh, D., Buggy McAfee update whacks Windows XP PCs, news.cnet.com/8301-1009_3-20003074-83.html, retrieved July 9, 2010.

[48] McGraw, G., Silver bullet talks with Matt Bishop, IEEE Security & Privacy, 7, (6), 6-10, 2008.

[49] McGraw, G., *Software Security: Building Security In*, Upper Saddle River, NJ: Addison-Wesley, 2006.

[50] Mead, N., Hough, E., Security requirements engineering for software systems: case studies in support of software engineering education, Proceedings of the 19th Conference on Software Engineering Education & Training, 149-158, 2006.

[51] Narasimhan, V., Das, M., Data and information security (DIS) for BS and MS programs: a proposal, ACM SIGCSE Bulletin, 42, (4), 95-99, 2008.

[52] Neumann, P., Risks to the public, ACM SIGSOFT software engineering notes, 25-37, 2006.

[53] North, M., North, M., North, S., Security from the bottom-up: compliance regulations and the trend toward design-oriented web applications, Journal of Computer Sciences in Colleges, 26, (4), 54-60, 2009.

[54] Null, L., Integrating security across the computer science curriculum, Journal of Computing Sciences in Colleges, 21, (5), 170-178, 2004.

[55] Pierce, R., Percentage difference vs percentage error, www.mathsisfun.com/data/percentage-difference-vs-error.html, retrieved September 25, 2010.

[56] Pothamsetty, V., Where security education is lacking, Proceedings of the 2nd annual conference on information security curriculum development, 54-58, 2005.

[57] Randall, R., The COLOSSUS, citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.5075&rep=rep1&type=pdf, retrieved July 9, 2010.

[58] Ranum, M., Security: the root of the problem, Queue, 3, (4), 44-49, 2004.

[59] Seacord , R., Secure coding in C and C++ of strings and integers, IEEE Security & Privacy, 5, (1), 74-76, 2006.

[60] Sexton, J., Establishing and undergraduate information assurance (information security) program at a small liberal arts college, Journal of Computing Sciences in Colleges, 26, (2), 234-240, 2008.

[61] Smarkusky, D., Smith, H., Team projects throughout the curriculum, Journal of Computing Sciences in Colleges, 21, (5), 119-129, 2004.

[62] Snyder, R., Ethical hacking and password cracking: a pattern for individualized security exercises, Proceedings of the 3rd annual conference on Information security curriculum development, 13-18, 2006.

[63] Sorkin, S., Tupper, D., Beiderman, A., Harmeyer, K., Mento, B., Creating an internet and multimedia technology program in a computer science department, Journal of Computing Sciences in Colleges, 20, (3), 32-44, 2003.

[64] Stallings, W., *Data and Computer Communications 8th Ed.*, Upper Saddle River, NJ: Prentice Hall, 2007.

[65] Stallings, W., Brown, L., *Computer Security: Principles and Practice*, Upper Saddle River, NJ: Prentice Hall, 2008.

[66] Stamat, M., Humphries, J., Training ≠ education: putting secure software engineering back in the classroom, Proceedings of the 14th Western Canadian Conference on Computing Education, 116-123, 2009.

[67] Stroustrup, B., Programming in an undergraduate CS curriculum, Proceedings of the 14th Western Canadian Conference on Computing Education, 82-89, 2009.

[68] Tanenbaum, A., *Computer Networks 3rd Ed.*, Upper Saddle River, NJ: Prentice Hall, 1996.

[69] Taylor, B., Azadegan, S., Moving beyond security tracks: integrating security in cs0 and cs1, Proceedings of the 39th SIGCSE technical symposium on computer science education, 320-324, 2008.

[70] Taylor, B., Azadegan, S., Threading secure coding principles and risk analysis into the undergraduate computer science and information systems curriculum, Proceedings of the 3rd annual conference on information security curriculum development, 24-29, 2006.

[71] Vaughn, R., Dampier, D., Warkentin, M., Building an information security education program, Proceedings of the 1st annual conference on information security curriculum development, 41-45, 2004.

[72] Weiss, R., Adding information assurance to the curriculum, Journal of Computing Sciences in Colleges, 24, (2), 46-48, 2006.

[73] Werner, L., Teaching principled and practical information security, Journal of Computing Sciences in Colleges, 22, (1), 81-89, 2004.

[74] West, R., The psychology of security, Communications of the ACM, 53, (4), 34-40, 2008.

[75] Whitman, M., Mattord, H., Designing and teaching information security curriculum, Proceedings of the 1st annual conference on information security curriculum development, 1-7, 2004.

[76] Wilson, B., Aman, J., Bourget, J., Wanted: trained security specialists, Journal of Computer Sciences in Colleges, 26, (2), 50-55, 2008.

[77] Wysopal, C., Nelson, L., Zovi, D., Dustin, E., *The Art of Software Security Testing*, Upper Saddle River, NJ: Symantec Press, 2007.

[78] Yang, A., Computer security and impact on computer science education, Journal of Computing in Small Colleges, 18, (4), 233-246, 2001.

[79] Yasinsac, A., McDonald, J., Foundations for security aware software development education, Proceedings of the 39th Hawaii International Conference on System Sciences, 219c-219c, 2006.

APPPENDICES

Appendix A: Informational Script on Human Subject Research

Informational Script on Human Subject Research

(to be read to the students prior to testing)

Professor Cain is conducting a research project towards his doctorate entitled Computer Science Education: Secure Software. The purpose of this research is to determine the effect of his Special Topics: Secure Software course on your knowledge of security.

Class time will be used to administer in-class pre-tests and post-tests respectively at the beginning and ending of the Spring 2010 semester. No personally identifiable information will be collected during this study. Your handwritten answers will be typed by SBU staff, at which point your original answer sheets will be shredded. Anonymous data will be computerized and encrypted for storage. Statistical results derived from the data will also be anonymous but may be published in the future by Professor Cain.

The records of this study will be kept private. Any written results will discuss group findings and will not include information that will identify you. Research records will be stored securely and only Professor Cain and individuals responsible for research oversight will have access to the records. It is possible that the consent process and data collection will be observed by research oversight staff responsible for safeguarding the rights and wellbeing of people who participate in research.

There are no known risks associated with this project which are greater than those ordinarily encountered in daily life.

Your participation in this is voluntary. You may withdraw from participation in the pre-test and/or the post-test at any time prior to turning in the test in question without any penalty or fear of reprisal. If you wish to withdraw, please tell me.

Do NOT put your name anywhere on the test. You will have until the end of this class to complete the test. When you complete the test, turn in the test to me, then go across the hall to the computer lab. Create a file on your personal, private F drive. Save the 6 digit pseudo-random number you find on the top of the first page of your pre-test information sheet in that file for use during the post-test.

Any questions?

Please tear off and read the first two pages of the packet you have been given at this time. They contain information about this study. You may retain these two pages. After you have read the first two pages, write the 6 digit pseudo-random number found on the top of the first page of your pre-test information sheet onto the top of your test, and begin the test.

Appendix B: Informational Handout on Human Subject Research

Information on Human Subject Research

Project Title: Computer Science Education: Secure Software

Investigators:

James Cain, Principal Investigator, Bachelor of Science in Electrical Engineering, Master of Science in Computer Science, Assistant Professor of Computer and Information Sciences at Southwest Baptist University, Ph.D. student in computer science at Oklahoma State University.

Purpose:

The purpose of this research is to determine the effect of his Special Topics: Secure Software course on your knowledge of security.

Procedures:

Class time will be used to administer in-class pre-tests and post-tests respectively at the beginning and ending of the Spring 2010 semester. You will be allowed 50 minutes to complete the pre-test and 50 minutes to complete the post-test.

Risks of Participation:

There are no known risks associated with this project which are greater than those ordinarily encountered in daily life.

Benefits:

You will be exposed to questions about computer security. These questions may instill curiosity to further your study of computer security.

Confidentiality:

No personally identifiable information will be collected during this study. Handwritten answers will be typed by SBU staff, at which point the original answer sheets will be shredded. Anonymous data will be computerized and encrypted for storage. Your pre-test results will be matched with your post-test results via the pseudo-random number written on the top of each test. Statistical results derived from the data will also be anonymous but may be published in the future by the principle investigator.

The records of this study will be kept private. Any written results will discuss group findings and will not include information that will identify you. Research records will be stored securely and only researchers and individuals responsible for research oversight will have access to the records. It is possible that the consent process and data collection will be observed by research oversight staff responsible for safeguarding the rights and wellbeing of people who participate in research.

Compensation:

Participation in this study cannot be compensated.

Contacts:

If you have questions about your rights as a research volunteer, you may contact:

James Cain, Principal Investigator, 107 Gene Taylor National Free Enterprise Center, Bolivar, MO 65613, 417-328-1680 or jcain@sbuniv.edu

Dr. John Murphy, SBU RRB Chair, Wheeler Science Center, Bolivar, MO 65613, 417-328-1494 or jmurphy@sbuniv.edu

Dr. Shelia Kennison, OSU IRB Chair, 219 Cordell North, Stillwater, OK 74078, 405-744-3377 or irb@okstate.edu

Dr. Blayne Mayfield, Advisor, 100 Telecom Center, Stillwater, OK 74078, 405-744-3471 or bem@cs.okstate.edu

Participant Rights:

Your participation in this is voluntary. You may withdraw from participation in the pre-test and/or the post-test at any time prior to turning in the test in question without any penalty or fear of reprisal. Returning your completed test indicates your willingness to participate in this research.

THIS PROJECT HAS BEEN REVIEWED BY THE SOUTHWEST BAPTIST UNIVERSITY RESEARCH REVIEW BOARD FOR RESEARCH AND RESEARCH-RELATED ACTIVITIES INVOLVING HUMAN SUBJECTS (417) 326-1659.

Appendix C: Survey Instrument

For the first six questions, circle the one <u>best</u> answer to each question.

1. What is wrong with the program shown below?

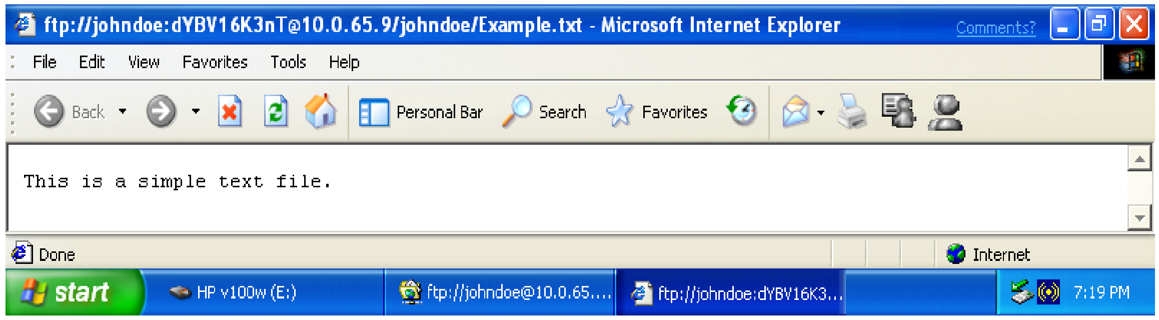
   A. Incorrect execution privilege level; program should not be run with administrative access to the entire system.

   B. Internet Explorer 6 has security updates available that have not yet been installed but should be installed.

   C. Internet Explorer 6 is obsolete and should be replaced with Internet Explorer 7 or 8.

   D. Information leakage.

   E. None of the above.



ftp://johndoe:dYBV16K3nT@10.0.65.9/johndoe/Example.txt - Microsoft Internet Explorer

```
This is a simple text file.
```

Comments?
File Edit View Favorites Tools Help
Back
Personal Bar
Search
Favorites
Done
Internet
start
HP v100w (E:)
ftp://johndoe@10.0.65....
ftp://johndoe:dYBV16K3...
7:19 PM

2. Sensitive data does not have to be encrypted if it is stored in a private account on a computer with up-to-date operating system, anti-virus, anti-spyware, and firewall software.

   A. True.

   B. False.

3. Which is the greater risk:

   A. A weak password.

   B. A strong password taped to the bottom side of the keyboard.

4. Can the `java.util.Random` class produce cryptographically strong random sequences?

   A. Yes.

B. Yes, if the seed can be kept secret.

C. Yes, if the seed can be kept secret and the attacker can never sample the generated sequence.

D. No.

5. Can "home-grown" cryptographic systems provide excellent security through their obscurity?

   A. Normally, yes.

   B. Normally, yes, if their keys and codebooks can be kept secure.

   C. Normally, yes, if their keys, codebooks, source code, and byte code can be kept secure.

   D. Possibly, but not normally.

6. Use cases should be created for:

   A. Every theoretically allowable use of the program.

   B. Every theoretically possible use of the program.

   C. Every realistically allowable use of the program.

   D. Every realistically possible use of the program.

For all remaining questions, circle <u>all</u> correct answers. There may be more than one correct answer for some questions. All java methods should be assumed to be in properly constructed classes with any required java libraries properly imported. The code provided does not contain comments because this is a test of code comprehension. Ignore issues of code style and focus on issues of code correctness.

7. Where in the software development process should code-defect risk analysis be performed?

   A. Stage 1: Problem Analysis and Use Case Design.

   B. Stage 2: Architectural and Algorithm Design.

   C. Stage 3: Code Implementation.

   D. Stage 4: Testing.

   E. Stage 5: Deployment.

   F. Stage 6: Maintenance.

8. Risk-based testing should be:

   A. Planned prior to coding.

   B. Performed regularly throughout project deployment and maintenance.

   C. More reliable than penetration testing.

   D. A valid substituted for penetration testing.

   E. None of the above.

9. Penetration testing:

   A. Requires expert hacking skills.

   B. Is best performed by hackers.

   C. Is better at finding security flaws than risk-based testing.

   D. None of the above.

10. What is the purpose of a code review?

   A. Keeping the team moving along the established timeline and insuring the waypoint production objectives are achieved.

   B. Verifying the code adheres to the design plans.

   C. Keeping the entire project team appraised on what others are doing on the project.

   D. Finding and fixing bugs.

   E. None of the above.

11. Are Java programs vulnerable to command injection attacks?

   A. No, because Java programs are compiled prior to execution.

   B. No, because Java programs are not interpreted but rather are run on a virtual machine.

   C. Yes, if they compile any code into byte code during their own execution.

   D. Yes, if they bring extra classes into execution at the same time on the same virtual machine.

   E. Yes, if user-specified data is sent to be interpreted by the operating system, the shell, or any interpreters.

12. Are Java programs vulnerable to format string attacks?

    A. Yes, if the code that translated the Java source code into Java byte code was written in a C-family language.

    B. Yes, if the Java Virtual Machine running the Java byte code was written in a C-family language.

    C. Yes, if the operating system was written in a C-family language.

    D. No.

13. Indicate which problem(s) exist in the `BubbleSort` method.

    A. Numeric overflow or underflow.

    B. Exception not caught or thrown.

    C. Ignored return value.

    D. Race condition.

    E. Failure to protect stored data.

    F. No problems exist in this code.

```
public boolean BubbleSort (int [] data)
{
  if (data != null)
  {
    for (int j = 0; j < data.length; j++)
    {
      for (int k = 0; k < (data.length - 1); k++)
      {
        if (data [k] < data [k + 1])
        {
          int temp = data [k];
          data [k] = data [k + 1];
          data [k + 1] = temp;
        }
      }
    }
    return true;
  }
  else
  {
    return false;
  }
}
```

14. Correct any problem(s) in the `BubbleSort` method.

15. Indicate which problem(s) exist in the `SearchForAllMatches` method. Assume that the `BubbleSort` method called in this problem is the one from problem 13.

A. Numeric overflow or underflow.

B. Exception not caught or thrown.

C. Ignored return value.

D. Race condition.

E. Failure to protect stored data.

F. No problems exist in this code.

```
public void SearchForAllMatches (int [] data, int target)
{
  BubbleSort (data);
  for (int j = 0; j < data.length; j++)
  {
    if (data [j] == target)
    {
      System.out.print ("Value " + target);
      System.out.println (" found at array location " + j);
    }
  }
}
```

16. Correct any problem(s) in the `SearchForAllMatches` method.

17. Indicate which problem(s) exist in the `Tally` method and class data member. Assume that the class it is in extends the `Thread` class.

    A. Numeric overflow or underflow.

    B. Exception not caught or thrown.

    C. Ignored return value.

    D. Race condition.

    E. Failure to protect stored data.

    F. No problems exist in this code.

```
public static int total;
public int Tally (int update)
{
  total = total + update;
  return total;
}
```

18. Correct any problem(s) in the `Tally` method and class data member.

Appendix D: Course Syllabus

CIS 2953 (Special Topics: Secure Software)

Required Text:
Howard, M., LeBlanc, D., Viega, J., *24 Deadly Sins of Software Security,* New York, NY: McGraw-Hill/Osborne, 2009. ISBN 978-0-07-162675-0

Recommended Text: (on reserve in the SBU Library)
McGraw, G., *Software Security: Building Security In,* Upper Saddle River, NJ: Addison-Wesley, 2006. ISBN 978-0-321-35670-3

Description:
A study of the most common security flaws in modern computer programs. Students will develop and refine secure programming techniques through correcting flaws in example programs and creation of their own secure programs.

Prerequisite:
A student is required to have completed Computer Science I.

TENTATIVE COURSE SCHEDULE

Week	Main Lecture Topics	Required Text	Recom. Text	Assignments
1/24	"The Security Problem"		Chapter 1	Pre-Test
1/31	Overflow & Underflow	Chapter 7	Chapter 4	Prog. 1 assigned
2/7	Exceptions	Chapter 9		
2/14	Errors	Chapter 11		Prog. 1 due, Prog. 2 assigned
2/21	Usability	Chapter 14	Chapter 5	
2/28	Information Leakage	Chapter 12		Prog. 2 due, Prog. 3 assigned
3/7	Data Protection	Chapter 17		
3/14	Buffers	Chapter 5	Chapter 8	Prog. 3 due, Prog. 4 assigned
3/21	Strings	Chapter 6		Midterm Exam
4/4	Command Injection	Chapter 10		
4/11	Race Conditions	Chapter 13	Chapter 7	Prog. 4 due, Prog. 5 assigned
4/18	Privilege	Chapter 16		
4/25	Passwords	Chapter 19		
5/2	Random Numbers	Chapter 20	Chapter 6	Prog. 5 due, Prog. 6 assigned
5/9	Cryptography	Chapter 21		Post-Test
5/16	Final Review			FINAL EXAM & Prog. 6 due

Course Goals:

Students who successfully complete this course will:

- Know the importance of security in modern computer systems.
- Understand that buggy code is exploitable code.
- Be familiar with 14 of the 24 most common categories of security-related programming flaws.
- Be able to spot instances of these 14 categories of flaws in existing Java programs and be able to fix them.
- Be able to write Java programs that do not contain instances of these 14 categories of flaws.

Organization:

- There will be three lecture periods of 50 minutes each per week.
- There will be 6 Java programming assignments.
- There will be 1 midterm exam and 1 comprehensive final exam.
- There will be 1 pre-test and 1 post-test.

Class Grading:

The grades for this course will be based on the following percentage breakdown:

• 60%	Programs
• 20%	Midterm Exam
• 20%	Final Exam
• 0%	Pre-Test
• 0%	Post-Test

Grading Scale:

Course grades will be determined by the following scale:

A	90% or greater
B	between 80% and 90%
C	between 70% and 80%
D	between 60% and 70%
F	below 60%

Program Submission:

Students are required to submit their source code for grading. Assignments submitted in executable form without source code will receive zero credit. Programs that fail to compile on the Taylor 221 lab machines will also receive zero credit.

Program Code Authoring:

Students are responsible for writing their own code. Using another individual's code without the prior consent of their professor, even when properly credited to the original author, will be considered to be plagiarism. Co-authoring of code with another individual will also be considered plagiarism. Students experiencing difficulty with coding assignments are directed to consult their professor for appropriate assistance.

Program 1:
Write a Java program that will accept integers from the command line, add them up, and print out their total to the screen. Make sure your code gets the correct total in light of potential arithmetic overflow and underflow.

Program 2:
Write a Java program that will accept integers from the command line, add them up, and print out their total to the screen. Make sure your code gets the correct total in light of potential arithmetic overflow and underflow. Make sure your code cannot be crashed due to any particular command line parameters, including non-integer values.

Program 3:
Correct the information leakage problem(s) in the Java code provided. Correct any other problems in the code and make sure your code cannot be crashed by any particular choices of user input, including obviously erroneous input.

Program 4:
Correct the buffer overflow problem(s) in the Java code provided. Correct any other problems in the code and make sure your code cannot be crashed by any particular choices of user input, including obviously erroneous input.

Program 5:
Correct the race condition problem(s) in the Java code provided. Correct any other problems in the code and make sure your code cannot be crashed by any particular choices of user input, including obviously erroneous input.

Program 6:
Write a Java class that will perform a XOR cipher encryption. In addition to a constructor and any other methods you need, your class should contain an `Encrypt` method and a `Decrypt` method. The `Encrypt` and `Decrypt` methods should each accept a single String class object parameter and should each return a String class object. Make sure your code cannot be crashed no matter what parameters are passed into your methods.

Appendix E: Revised Course Schedule

REVISED COURSE SCHEDULE

Week	Main Lecture Topics	Required Text	Recom. Text	Assignments
1/24	"The Security Problem"		Chapter 1	Pre-Test
1/31	Overflow & Underflow	Chapter 7	Chapter 4	Prog. 1 assigned
2/7	Exceptions & Errors	Chapter 9 & 11	Chapter 5	
2/14	Prog. 1 Code Review			Prog. 1 due, Prog. 2 assigned
2/21	Usability & Leakage	Chapt. 12 & 14		
2/28	Prog. 2 Code Review	Chapter 12		Prog. 2 due, Prog. 3 assigned
3/7	Data Protection	Chapter 17		
3/14	Buffers & Prog. 3 Code Review	Chapter 5	Chapter 8	Prog. 3 due, Prog. 4 assigned
3/21	Strings	Chapter 6		Midterm Exam
4/4	Race Conditions	Chapter 13	Chapter 7	
4/11	Command Injection & Prog. 4 Code Review	Chapter 10		Prog. 4 due, Prog. 5 assigned
4/18	Privilege	Chapter 16		
4/25	Random Numbers	Chapter 20		
5/2	Cryptography & Prog. 5 Code Review	Chapter 21	Chapter 6	Prog. 5 due, Prog. 6 assigned
5/9	Passwords	Chapter 19		Post-Test
5/16	Final Review			FINAL EXAM & Prog. 6 due

Appendix F: Program 3 Starter Code

```
// Program 3 starter code
// Starter Code By: James Cain (jcain@sbuniv.edu)
// This is the required starter code for this assignment.
// Assignment Description:
// Read the following Java API descriptions:
// java.sun.com/javase/7/docs/api/java/util/Random.html
// java.sun.com/javase/7/docs/api/java/io/File.html
// java.sun.com/javase/7/docs/api/java/util/Scanner.html
// java.sun.com/javase/7/docs/api/java/io/PrintStream.html
// java.sun.com/javase/7/docs/api/java/lang/String.html
// java.sun.com/javase/7/docs/api/java/lang/Math.html
// Correct the information leakage problem(s) in the Java code
// provided.  Correct any other problems in the code and make
// sure your code cannot be crashed by any particular choices of
// user input, including obviously erroneous input.
// Algorithmic Description:
// Whoever wrote this program should have told you what it does.
// If you really cannot understand what this code does and
// therefore cannot do the assignment, ask your professor to
// explain it to you and he will.
// Last comment: Don't you wish this wasn't the last comment?

import java.util.Random;
import java.io.File;
import java.util.Scanner;
import java.io.PrintStream;
import java.lang.Math;

public class cis2953sp10p3 {

public static void main (String [] args) throws Exception{
  Random rand = new Random();
  File infile = new File (args [0]);
  File outfile = new File ("PasswordList.txt");
  Scanner kbd = new Scanner (System.in);
  Scanner Infile = new Scanner (infile);
  PrintStream Outfile = new PrintStream (outfile);
  String uppers = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  String lowers = "abcdefghijklmnopqrstuvwxyz";
  String digits = "0123456789";
  String symbols = "`~!$%^&*()_+{}|:<>?/.,';][=-";
  String total = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
                 "abcdefghijklmnopqrstuvwxyz" +
                 "0123456789`~!$%^&*()_+{}|:<>?/.,';][=-";
  outfile.setReadable (true, true);
  System.out.print ("Minumum uppercase: ");
  int Uppers = kbd.nextInt ();
  System.out.print ("Minumum lowercase: ");
  int Lowers = kbd.nextInt ();
  System.out.print ("Minumum digits: ");
  int Digits = kbd.nextInt ();
  System.out.print ("Minumum symbols: ");
  int Symbols = kbd.nextInt ();
  System.out.print ("Minumum total characters: ");
  int Total = kbd.nextInt ();
  char [] password = new char [Total];
  String user;
  int i,j,k;
  char temp;
  while (Infile.hasNext()) {
    user = Infile.next();
    for (i = 0; i < Uppers; i ++) {
      password [i] = uppers.charAt (Math.abs (
                     rand.nextInt ()) % 26);
    }
    for (i = Uppers; i < (Uppers + Lowers); i ++) {
      password [i] = lowers.charAt (Math.abs (
                     rand.nextInt ()) % 26);
    }
    for (i = Uppers + Lowers;
         i < (Uppers + Lowers + Digits); i ++) {
      password [i] = digits.charAt (Math.abs (
                     rand.nextInt ()) % 10);
    }
    for (i = Uppers + Lowers + Digits;
         i < (Uppers + Lowers + Digits + Symbols); i ++) {
      password [i] = symbols.charAt (Math.abs (
                     rand.nextInt ()) % 26);
    }
    for (i = Uppers + Lowers + Digits + Symbols;
         i < Total; i ++) {
      password [i] = total.charAt (Math.abs (
                     rand.nextInt ()) % 88);
    }
    for (i = 0; i < 1000; i ++) {
      j = Math.abs (rand.nextInt ()) % Total;
      k = Math.abs (rand.nextInt ()) % Total;
      temp = password [j];
      password [j] = password [k];
      password [k] = temp;
    }
    System.out.println (user + " " +
                        (new String (password)));
    Outfile.println (user + " " + (new String (password)));
  }
}
}
```

Appendix G: Program 4 Starter Code

```
// Program 4 starter code
// Starter Code By: James Cain (jcain@sbuniv.edu)
// This is the required starter code for this assignment.
// Correct the buffer overflow problem(s) in the Java code
// provided.  Correct any other problems in the code and make
// sure your code cannot be crashed by any particular choices of
// user input, including obviously erroneous input.
// Do not simply replace all of the character arrays with Java
// Strings.  Furthermore, you are not permitted to use Java
// Strings anywhere in this assignment.
// Create a main method to test your code.

public class cis2953sp10p4 {

public char [] strcat (char [] destination, char [] source) {
// This method concatenates the null-character terminated
// characters within the destination character array with the
// characters from the null-character terminated characters in
// the source character array and returns the destination
// character array.  The behavior is exactly like the behavior of
// the C-language function "strcat".  By correcting this code,
// the student should gain an understanding of exactly what
// C-language does wrong in the function "strcat" and similar
// functions in the string.h and stdio.h libraries.

  // Find the integer value i such that the i-th character
  // of the destination array is the first null character
  int i = 0;
  while (destination [i] != '\0') {
    i++;
  }  // while (destination [i] != '\0') {...
  // Copy the source array into the destination array,
  // overwriting the existing characters beginning at the
  // first null character within the destination array
  int j = 0;
  while (source [j] != '\0') {
    destination [i] = source [j];
    i++;
    j++;
  }  // while (source [j] != '\0') {...
  // Null-terminate the destination array after the last
  // character concatenated into it from the source array
  destination [i] = '\0';
  // Return the destination array
  return destination;
}  // public char [] strcat (char [] destination,
   //   char [] source) {...
}  // public class cis2953sp10p4 {...
```

Appendix H: Program 5 Starter Code

```
// Program 5 starter code
// Starter Code By: James Cain (jcain@sbuniv.edu)
// This is the required starter code for this assignment.

// Correct the race condition problem(s) in the Java code
// provided. Correct any other problems in the code and make sure
// your code cannot be crashed by any particular choices of user
// input, including obviously erroneous input.

public class RetirementAccount {
  private float Balance;
  private float InterestRate;

  public RetirementAccount () {
    Balance = (float) 0;
    InterestRate = (float) 0;
  } // public RetirementAccount () {...

  public float GetBalance () {
    return Balance;
  } // public float GetBalance () {...

  public float GetRate () {
    return InterestRate;
  } // public float GetRate () {...

  public void Debit (float Amount) {
    Balance = Balance - Amount;
  } // public void Debit (float Amount) {...

  public void Credit (float Amount) {
    Balance = Balance + Amount;
  } // public void Credit (float Amount) {...

  public void ApplyFee (float Amount) {
    Balance = Balance - Amount;
  } // public void ApplyFee (float Amount) {...

  public void ApplyInterest () {
    Balance = Balance + (Balance * InterestRate);
  } // public void ApplyInterest () {...

  public void SetRate (float Rate) {
    InterestRate = Rate;
  } // public void SetRate (float Rate) {...
} // public class RetirementAccount {...
```

Appendix I: Survey Data

The tables in Appendix I are laid out with a numbered list of subjects corresponding to the columns and the possible multiple choice answers and coding problem numbers corresponding to the rows. Each possible multiple choice answer is recorded as either selected (denoted 1) or not selected (denoted 0). Coding problems 14, 16, and 18 are recorded as either correct (denoted 1) or incorrect (denoted 0). Problems 16* and 18* are coding problems 16 and 18 rerecorded as either having any appairent *attempt* to do the problem (denoted 1) or not having any appairent *attempt* to do the problem (denoted 0).

Tables I.1 and I.3 are numbered to show the pairing for control subjects that took both the pre-test and the post-test. Likewise, tables I.2 and I.4 are numbered to show the pairing for test subjects that took both the pre-test and the post-test. The answer key and the tests that could not be properly paired beween pre-test and post-test are listed in table I.5. The columns in table I.5 are labeled for the answer key (denoted K, for “Key”), control subject pre-tests that could not be paired with post-tests (denoted C, for “Control”), test subject pre-tests that could not be pair with post-tests (denoted T, for “Test”), and post-tests that could not be paired with pre-tests (denoted P, for “Post”).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1A	0	1	0	0	1	0	1	1	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0
1B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1C	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1
1D	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0
1E	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	1	0	1	0	1	0
2A	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0
2B	1	0	1	1	1	0	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0	1
3A	0	1	0	1	1	0	1	1	0	0	0	0	0	0	1	1	0	1	0	0	1	0	1	0
3B	1	0	1	0	0	1	0	0	1	1	1	1	1	1	0	0	1	0	1	1	0	1	0	1
4A	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4B	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1
4C	0	1	0	0	1	1	1	0	1	1	1	1	0	1	0	1	1	1	0	1	1	0	0	0
4D	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0
5A	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
5B	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	1	1
5C	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	1	1	0	0	0	0	1	0	0
5D	0	1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0
6A	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
6B	0	0	0	0	0	1	0	1	0	0	1	1	1	1	0	1	1	0	1	1	1	1	1	1
6C	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6D	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
7A	0	1	1	0	0	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	1	0	0
7B	0	1	0	0	0	0	1	1	1	0	1	0	0	1	0	0	0	0	0	1	1	1	0	1
7C	1	1	0	0	0	0	1	1	0	1	1	0	1	1	0	1	1	1	0	1	0	0	1	1
7D	1	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	0	0	0	0	0	1	0
7E	0	1	0	0	0	0	0	1	0	0	0	1	0	1	0	1	1	0	0	0	0	0	0	0
7F	1	0	1	0	0	1	1	1	0	1	0	1	0	0	0	1	1	0	0	0	0	1	0	1
8A	0	1	0	0	1	1	1	1	0	0	0	0	1	0	1	0	1	1	1	1	1	1	0	1
8B	1	0	1	1	1	1	1	1	0	1	0	1	0	1	0	1	1	1	1	1	0	1	1	0
8C	0	0	0	0	0	0	0	1	1	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0
8D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9A	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0
9B	0	1	0	0	1	0	1	1	0	0	1	0	0	0	0	1	0	0	0	1	1	1	0	0
9C	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	0	1	1	1
9D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10A	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0
10B	1	1	1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	0	1	1	0	1	1	1
10C	1	0	0	1	0	0	0	1	0	1	0	0	1	1	0	0	1	1	0	1	0	1	1	0
10D	0	1	1	0	1	1	1	0	0	1	1	0	0	1	0	0	0	1	1	1	1	1	1	1
10E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11A	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0
11B	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0
11C	0	0	0	0	1	0	1	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	1
11D	0	1	1	0	0	1	1	0	0	1	1	0	1	0	0	0	0	1	1	0	0	0	0	0
11E	1	1	0	1	1	1	1	1	1	1	0	1	0	0	0	1	1	1	0	0	1	1	0	0
12A	0	0	0	0	1	1	1	1	0	1	1	0	0	1	0	0	1	1	0	0	1	0	0	0
12B	1	0	0	0	1	0	0	1	1	1	1	0	1	0	0	0	0	0	1	1	0	0	1	0
12C	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1
12D	0	1	1	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
13A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
13B	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
13C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
13D	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
13E	0	0	1	0	1	0	0	0	1	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0
13F	1	1	0	1	0	0	1	1	0	1	1	0	1	0	1	0	1	0	1	0	0	1	1	1
14	1	1	0	1	0	0	1	1	0	1	1	0	1	0	1	0	1	0	1	0	0	1	1	1

Table I.1 (part a) Pre-Test Control Subject Data

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
15A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
15B	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0
15C	0	1	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
15D	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15E	0	0	1	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0
15F	1	0	0	1	0	1	0	1	1	0	1	1	0	0	1	1	0	0	0	0	0	1	1	1
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17A	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
17B	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
17C	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17D	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
17E	1	0	1	1	0	0	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	0	1	0
17F	0	1	0	0	0	1	1	1	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	1
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16*	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
18*	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	0	0	0	0	0

Table I.1 (part b) Pre-Test Control Subject Data

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1A	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1B	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1C	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	1	1	1
1D	1	1	1	0	0	0	0	1	1	0	1	1	1	0	1	0	1	0	0	0	0
1E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2B	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3A	0	1	0	0	0	0	0	1	1	0	1	1	1	1	1	1	1	0	1	0	1
3B	1	0	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0
4A	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
4B	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
4C	1	0	1	1	1	1	1	0	0	1	0	1	0	1	1	1	0	1	1	0	0
4D	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	1
5A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
5B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5C	1	0	0	1	1	0	1	0	0	1	1	0	1	1	1	0	0	0	1	1	0
5D	0	1	1	0	0	1	0	1	1	0	0	1	0	0	0	1	1	1	0	0	0
6A	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
6B	1	1	0	0	0	1	1	1	1	1	0	0	0	0	1	1	0	1	0	0	0
6C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6D	0	0	0	0	1	0	0	0	0	0	0	1	1	1	0	0	1	0	0	1	1
7A	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0
7B	0	1	1	1	0	1	0	0	1	0	0	0	1	0	1	0	1	0	0	0	1
7C	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	1	1	1	1	1
7D	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	0	1	0	1	0	1
7E	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1
7F	0	1	1	0	0	0	0	0	1	0	1	0	1	1	0	1	1	1	0	0	1
8A	0	1	0	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	0	0	0
8B	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	1	1
8C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
9A	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9B	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	1	1	0	0	0
9C	1	1	1	1	0	0	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1
9D	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0

Table I.2 (part a) Pre-Test Test Subject Data

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
10A	1	0	1	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0
10B	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0
10C	1	0	1	0	1	0	0	1	1	0	0	0	0	1	1	0	1	1	0	1	1
10D	1	1	1	1	1	1	1	0	0	1	1	0	0	1	1	1	1	1	1	0	1
10E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11A	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0
11B	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
11C	1	0	0	0	0	1	0	0	1	0	1	0	0	0	1	0	1	0	1	0	1
11D	0	1	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0
11E	1	1	0	0	0	1	1	0	1	0	0	1	1	1	1	1	1	0	1	1	0
12A	0	0	1	0	1	1	1	1	0	0	0	0	1	1	1	0	0	1	0	1	0
12B	0	0	0	1	0	0	0	0	0	1	1	0	1	0	1	1	0	1	0	0	1
12C	1	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
12D	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0
13A	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0
13B	0	1	0	1	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	1
13C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
13D	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
13E	1	1	1	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
13F	0	0	0	0	0	1	1	0	0	0	1	0	0	1	1	1	1	1	0	0	0
14	0	0	0	0	0	1	1	0	0	0	1	0	0	1	1	1	1	1	0	0	0
15A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15B	0	1	1	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1
15C	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0
15D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15E	1	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
15F	0	0	0	0	0	0	1	0	1	0	1	0	0	1	1	1	0	1	1	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17A	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0
17B	0	1	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
17C	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
17D	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
17E	1	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	1
17F	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	1	0	1	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16*	1	0	1	1	1	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
18*	1	1	0	0	1	1	0	0	1	0	1	1	0	1	0	0	1	0	1	1	0

Table I.2 (part b) Pre-Test Test Subject Data

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1A	0	0	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	1	0	0	0	0	1	0
1B	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1C	1	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	0	1	1	0	1
1D	0	0	1	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0
1E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
2A	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0
2B	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	0	1
3A	0	1	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	1	0	0	1	1	1	0
3B	1	0	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	0	0	0	1
4A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4B	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
4C	1	1	1	1	1	1	1	0	1	0	1	0	1	1	0	1	1	1	1	1	1	0	1	0
4D	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0
5A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5B	0	1	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	0	0	1
5C	1	0	0	1	0	1	0	1	1	1	0	1	0	1	0	1	1	1	0	0	0	1	1	0
5D	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0

Table I.3 (part a) Post-Test Control Subject Data

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
6A	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
6B	0	0	0	1	0	1	1	1	0	0	1	1	1	1	1	1	0	1	1	0	0	1	0	1
6C	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6D	1	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	1	0	0	0
7A	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	1	1	0	0
7B	0	1	0	1	0	1	0	1	0	0	0	1	0	0	0	1	1	0	0	1	1	1	0	1
7C	1	1	1	1	0	1	1	1	0	1	1	0	1	1	1	1	0	1	1	0	1	1	1	0
7D	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	0	1	1	0	1	1	0	0
7E	1	0	1	0	0	0	0	1	0	1	0	0	0	0	0	1	1	0	0	0	1	0	0	0
7F	1	0	0	0	0	0	1	1	0	1	0	1	0	1	0	1	1	0	0	0	1	1	0	0
8A	0	1	1	1	1	1	1	1	0	1	1	1	0	1	1	0	0	0	0	0	1	0	0	1
8B	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	0	1	1	1	1	0
8C	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
8D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
9A	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	0
9B	1	1	0	0	1	1	1	1	0	0	0	0	0	1	0	0	0	1	0	1	0	1	0	0
9C	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	0	1	1	1
9D	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10A	0	1	0	0	1	0	1	1	0	1	0	0	0	0	0	0	0	1	0	0	1	0	1	0
10B	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	0	1	0	1
10C	1	1	1	0	1	0	1	1	0	1	0	0	0	0	0	0	1	1	1	0	0	1	0	0
10D	0	0	1	1	1	1	1	1	0	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0
10E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11A	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	0
11B	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11C	0	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	1	1
11D	0	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0
11E	1	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	1	0	0
12A	0	0	0	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	1	0	0	0	0	0
12B	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	1	0	0	0	1
12C	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0
12D	1	0	0	1	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	0
13A	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	1	0	1	0	0	0	1	0	0
13B	0	0	1	0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
13C	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13D	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13E	0	0	0	1	1	0	1	1	0	1	0	0	1	0	0	0	0	0	0	1	0	0	1	1
13F	1	1	0	0	0	0	0	0	0	0	1	1	0	1	1	0	1	0	1	0	1	0	0	0
14	1	1	0	0	0	0	0	0	0	0	1	1	0	1	1	0	1	0	1	0	1	0	0	0
15A	1	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	0	0	0	1	0	0	0
15B	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
15C	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15D	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
15E	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	1
15F	0	0	0	0	0	1	1	0	1	0	1	1	1	0	0	0	1	1	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17A	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
17B	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17D	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
17E	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	0	0	1	0
17F	0	0	0	1	0	1	1	0	1	0	1	1	0	0	1	0	1	1	0	1	1	1	0	1
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16*	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0
18*	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0

Table I.3 (part b) Post-Test Control Subject Data

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1A	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1C	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
1D	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0
1E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
2B	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1
3A	1	1	0	1	0	1	0	1	1	0	1	1	1	1	0	0	0	0	0	0	1
3B	0	0	1	0	1	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0
4A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4C	0	0	0	1	1	0	1	0	1	0	0	1	0	0	1	0	1	1	1	0	0
4D	1	1	1	0	0	1	0	1	0	1	1	0	1	1	0	1	0	0	0	1	1
5A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5B	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
5C	0	0	1	1	1	0	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0
5D	1	1	0	0	0	1	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0
6A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6B	1	1	0	1	0	1	1	1	1	1	0	1	0	1	1	1	0	1	1	0	0
6C	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
6D	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	1	0	0	1	0
7A	0	1	1	1	0	0	0	1	1	0	0	0	1	1	1	0	1	1	0	1	1
7B	0	1	1	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	1	1
7C	0	1	1	1	1	0	1	1	0	1	1	1	1	1	0	1	1	1	1	0	1
7D	1	0	1	1	1	0	0	1	0	1	1	0	1	1	0	1	1	1	0	0	1
7E	0	0	1	1	0	0	0	1	0	0	1	0	0	1	0	0	1	1	0	0	1
7F	1	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	1	1	0	0	1
8A	0	1	1	0	1	1	0	1	1	1	1	0	0	1	0	0	1	1	0	1	1
8B	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
8C	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0
8D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9A	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0
9B	1	1	0	1	0	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1
9C	1	0	1	0	1	0	1	0	0	1	1	0	1	1	1	1	0	1	1	1	1
9D	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
10A	0	1	0	0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	0	0	0
10B	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	1	1
10C	1	1	1	0	1	0	0	1	0	1	1	0	1	0	1	1	1	1	0	1	1
10D	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
10E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11A	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
11B	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0
11C	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	1	1
11D	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	1	0	0	0	1
11E	0	0	0	1	1	1	1	0	1	0	1	1	0	1	0	1	1	0	0	0	0
12A	1	0	0	0	1	1	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0
12B	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0
12C	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
12D	0	1	1	0	0	0	0	1	1	0	0	1	1	1	0	1	1	0	1	0	1
13A	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
13B	1	1	1	1	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	1	1
13C	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
13D	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	0	0
13E	0	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
13F	0	0	0	0	0	1	0	1	1	0	1	0	0	1	0	1	1	0	0	0	0
14	0	0	0	0	0	1	0	1	1	0	1	0	0	1	0	1	1	0	0	0	0

Table I.4 (part a) Post-Test Test Subject Data

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
15A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15B	1	1	0	0	1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	0	1
15C	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
15D	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
15E	1	1	0	0	1	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0
15F	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	1	0	1	0	1	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
17A	0	1	0	0	1	1	0	0	1	1	0	0	0	0	0	1	1	1	0	0	0
17B	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17D	0	1	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0
17E	1	1	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
17F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16*	1	1	1	0	1	0	0	0	0	1	1	1	0	1	1	0	1	0	0	0	0
18*	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0

Table I.4 (part b) Post-Test Test Subject Data

	K	C	C	C	C	C	C	C	T	T	P	P	P	P	P
1A	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1B	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0
1C	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0
1D	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1
1E	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0
2A	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
2B	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
3A	1	0	0	0	1	0	0	0	0	1	0	1	0	0	0
3B	0	1	1	1	0	1	1	1	1	0	1	0	1	1	1
4A	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0
4B	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
4C	0	0	1	0	0	1	0	0	0	1	0	0	0	0	1
4D	1	0	0	0	0	0	0	1	1	0	1	1	0	1	0
5A	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
5B	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
5C	0	0	0	0	0	1	1	0	1	0	0	0	1	0	0
5D	1	0	1	0	1	0	0	1	0	1	1	1	0	0	1
6A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6B	1	1	1	0	1	0	1	0	1	1	1	1	1	1	0
6C	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0
6D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
7A	1	0	0	0	0	0	1	0	0	0	1	1	0	0	0
7B	1	0	0	0	1	0	0	0	1	1	1	1	0	0	0
7C	0	1	1	1	0	1	1	0	0	0	1	1	0	0	1
7D	1	1	0	1	1	1	0	1	0	1	1	0	1	1	1
7E	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0
7F	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0
8A	1	0	1	0	0	1	0	1	1	1	0	1	1	0	1
8B	0	1	0	1	1	1	1	1	0	1	1	1	0	1	1
8C	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
8D	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
8E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9A	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0
9B	0	1	0	0	0	0	1	0	0	1	0	1	0	0	1
9C	0	0	0	0	1	1	0	1	1	0	1	1	1	1	0
9D	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0

Table I.5 (part a) Answer Key and Unpaired Subject Data

	K	C	C	C	C	C	C	C	T	T	P	P	P	P	P
10A	0	1	1	1	0	0	0	1	0	0	1	1	0	0	0
10B	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
10C	0	0	0	1	0	1	0	1	0	0	1	0	0	0	1
10D	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1
10E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11B	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
11C	1	1	0	0	0	1	0	1	0	0	1	0	0	0	0
11D	1	1	1	0	0	1	1	1	1	0	0	0	0	0	0
11E	1	0	1	1	1	1	0	0	0	0	0	1	1	1	0
12A	0	1	0	0	0	1	0	0	0	0	1	1	0	0	1
12B	0	1	0	1	0	1	0	0	0	0	1	1	1	0	0
12C	0	1	0	0	0	0	1	0	1	0	1	1	0	0	0
12D	1	0	1	0	1	0	0	1	0	1	0	0	0	1	0
13A	0	1	1	1	0	0	0	1	0	1	0	0	0	1	0
13B	0	1	0	1	0	0	0	0	0	0	1	0	1	0	0
13C	0	0	1	0	0	0	0	0	1	0	1	0	0	1	0
13D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13E	0	0	0	1	1	0	1	0	0	1	0	1	1	0	0
13F	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1
14	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1
15A	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
15B	1	1	0	1	0	0	0	0	1	0	1	0	0	0	0
15C	1	1	0	0	0	0	0	0	0	0	1	0	0	1	0
15D	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
15E	0	0	0	1	0	0	1	0	0	0	0	0	1	0	1
15F	0	0	1	0	1	1	0	1	0	0	1	0	0	0	0
16	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17A	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
17B	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0
17C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17D	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
17E	1	0	0	1	0	0	1	1	1	0	0	0	1	1	0
17F	0	1	0	0	1	1	0	0	0	1	1	1	0	0	0
18	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16*	1	1	0	0	0	0	0	0	1	1	0	1	0	0	0
18*	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0

Table I.5 (part b) Answer Key and Unpaired Subject Data

VITA

James Francis Cain III

Candidate for the Degree of

Doctor of Philosophy

Thesis: COMPUTER SCIENCE EDUCATION: SECURE SOFTWARE

Major Field: Computer Science

Biographical:

Education:

Completed the requirements for the Doctor of Philosophy in Computer Science at Oklahoma State University, Stillwater, Oklahoma in December, 2010.

Completed the requirements for the Master of Science in Computer Science at University of Missouri – Rolla, Rolla, Missouri in 1999.

Completed the requirements for the Bachelor of Science in Electrical Engineering at University of Missouri – Rolla, Rolla, Missouri in 1996.

Experience:

Southwest Baptist University, Bolivar, Missouri: Instructor, 1999 – 2008,
Assistant Professor, 2008 – present.

Professional Memberships:

ACM & ACM Special Interest Group on Computer Science Education
IEEE & IEEE Computer Society
National Society of Professional Engineers
Upsilon Pi Epsilon Computer Science Honor Society
Consortium for Computing Sciences in Colleges

Professional Meetings Attended:

ACM Technical Symposium on Computer Science Education 2001, 2005-2008
Consortium for Computing Sciences in Colleges, Central Plains Conference
2000-2002, 2006-2010

Name: James Francis Cain III Date of Degree: December, 2010

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: COMPUTER SCIENCE EDUCATION: SECURE SOFTWARE

Pages in Study: 119 Candidate for the Degree of Doctor of Philosophy

Major Field: Computer Science

Scope and Method of Study: Computer Science Education Human Subject Research

Findings and Conclusions:

Computer security problems have been increasing significantly as the Internet has been increasing the means to both access and to distribute both code and data. Attempts to address these problems through computer science education by focusing on information security, network security, and system security have not been entirely successful. The security problems are serious enough at this time that both industry and academia are looking for other solutions and even for other partial solutions. One of these proposed partial solutions focuses the security investigations on the commonality that underlies all software: code.

The author proposed that all computer science undergraduates should be required to take a computer security course that focuses on code security early in their undergraduate program. The objectives of this course would be to teach the importance of code security, to instruct in practical coding techniques for making programs more secure, and to provide practice in these secure coding techniques.

The author has taught an introductory security course with emphasis in code security over the course of one semester during this research project. The students in the course ranged from second semester freshman, straight out of Computer Science I, to seniors graduating at the end of that semester. While results from the pre-test and post-test surveys completed by course subjects were mixed, they suggested that the course was at least partially successful. The students did seem to have a better understanding of computer security but seem to have not improved as much within the area of secure coding as the author had anticipated. The author feels that more repetition and feedback on the writing of secure code will improve the course the next time it is offered.

The author believes these proposals are not a perfect solution for the present computer security problem. However, the author does believe that these proposals are a valid partial solution.

ADVISER'S APPROVAL: Dr. Blayne E. Mayfield