

## UC Davis

### UC Davis Previously Published Works

#### Title

Practice, Practice, Practice ... Secure Programmer!

#### Permalink

<https://escholarship.org/uc/item/1f96q5c3>

#### Authors

Dark, Melissa  
Belcher, Steven  
Bishop, Matt  
[et al.](#)

#### Publication Date

2015-06-01

Peer reviewed

# Practice, Practice, Practice... Secure Programmer!

**Melissa Dark:** Purdue University

**Stephen Belcher:** National Security Agency

**Ida Ngambeki:** Purdue University

**Matt Bishop:** University of California Davis

## I. Abstract

One of the major weaknesses in software today is the failure to practice defensive or secure programming. Most training programs include only a shallow introduction to secure programming, and fail to integrate and emphasize its importance throughout the curriculum. The addition of an ongoing, practical, mentored "clinic" or Secure Programming Clinic (SPC) is one way of addressing this lack without adding significantly to an already stretched curriculum. In order to properly design this clinic, it is important to identify the knowledge, skills and abilities (KSAs) needed to develop effective programmers. This paper describes the results of a Delphi Study undertaken to determine the primary knowledge areas in secure programming.

## II. Background

The current state of software today is generally poor. The quality of the software in most systems does not support the trust placed in that software. Software security failures are common, and the effects range from inconvenience to severe problems. For example, failing to properly handle an error condition made Facebook inaccessible for a few hours [John10]; the iPhone failed to ring in the New Year in 2010 [Bilt11]; a flaw in an election system used to count votes resulted in some votes from a precinct not being counted [Zett08]; a 2010 FDA study of recalled infusion pumps reported that one of the most common reported problems was software defects [FDA10]; and scientists analyzing on-board computers that control automobiles were able to exploit software vulnerabilities to control the car without physical access [CCK+11]. Indeed, much of computer security deals with handling problems created by programming errors, including analyzing and countering attacks that exploit these problems.

A variety of causes underlie this phenomenon. One cause is the failure of practitioners to practice "defensive programming," in which basic principles of robust coding guard against unexpected inputs and events. This style of programming is often called "secure programming" or "secure coding," but this term is a misnomer. By rights, it should refer to programming designed to satisfy a specified security policy—the definition of "secure", after all, is "satisfying a security policy" [Bish02]. But the term is used more broadly to refer to programming designed to prevent problems that might cause security breaches. This style of programming deals with common programming errors (such as failing to check that the size of an input is no greater than the size of where it is to be stored), and should more properly be called "robust programming." In this paper, we use the term secure programming to include both.

The failure of practitioners to practice this style is not due to incompetence. One factor stems from the market economy; those who do know how to practice secure programming are often not given the opportunity to do so because it would increase cost or time to market. Few believe that customers will accept longer delivery times, or higher prices, for more robust programs. Another factor is simply lack of preparation. As Evans and Reeder noted [EvRe10], "We ... [have a] desperate shortage of people who can design secure systems, write safe computer code, and create the ever more sophisticated tools needed to prevent, detect, mitigate and reconstitute systems from damage due to system failures and malicious

acts.” This problem is one of both quality and quantity, of ensuring that students get a sufficient amount of secure programming knowledge and practice in the curriculum, as well as ensuring that a sufficient number of students get it.

Critical questions then are: 1) what is the current state of secure programming, and 2) what knowledge skills and abilities are needed to develop effective programmers.

### III. The State of Teaching Secure Programming

The lack of secure programming is not a new phenomenon. Indeed, in 1971, the psychologist Gerald Weinberg codified it as his second law of programming [Wein71]: “If builders built buildings the way programmers wrote programs, then the first woodpecker to come along would destroy civilization.”

Most educational institutions do not change this situation. In introductory classes, students learn to write well-structured programs. They might learn to check for common errors such as array references being out of bounds or integers not in particular ranges. Further, this style of programming may affect their grades, a (sometimes large) portion of which depends on good style and error checking. But introductory textbooks do not explore this type of defensive programming in depth, leaving it to the instructor to cover the practice of secure programming using supplementary material that they develop or find [NB12].

The situation worsens in more advanced classes. The focus of these classes is on the principles and applications being covered, and programs are graded with a focus on how well they demonstrate knowledge of those concepts and applications. The grading rarely includes good programming style; the test is; if the code works, and if it exhibits an understanding of the data structures, algorithms, or other material covered in the class. Hence the practice of defensive programming atrophies through disuse. When the student graduates and enters the field of software engineering, the situation continues – the focus is on product development to meet schedules and features. Minimizing time to market comes at a cost, usually of the robustness and security of the product. Indeed, Palmer attributes the poor state of software to the three “bad habits” [Palm04, p. 10]: fast development, add-on security, and feature creep. Because time to market is critical, security is often added as an afterthought, as is making the software robust (a key component of security). Similarly, adding more features (called “feature creep”) makes the product more marketable, and thus is emphasized.

As an example of the criticality of this problem, consider a buffer overflow in a widely used cryptographic library, RSAREF2, found in 1999. It enabled the compromise of many security-based programs [CERT99, Core99]. The recent Heartbleed flaw found in OpenSSL is another example [Schn14]. OpenSSL is network software that secures Internet connections. It underlies much of the web-based purchasing mechanisms today, because those mechanisms assume an authenticated, encrypted, integrity-checked connection. The SSL protocol provides this. OpenSSL is a free implementation of this protocol, and is very widely used. Unfortunately, a failure to check bounds enabled an attacker to obtain sensitive information from a server (or client) running OpenSSL. The flaw was easy to fix, once found. The point is, secure systems rely upon libraries and other middleware to provide the security mechanisms, and failures in those mean the systems that rely on them are also not secure.

Correcting this situation requires a concerted effort to change the computing ecosphere — the marketplace, the development processes, and teaching. Focusing on the last, students should practice robust, defensive programming throughout their educational career. Unfortunately, this is easier said than done. First, many faculty lack experience in this style of programming because their expertise lies in other realms such as theory or specific aspects of systems, and so they have little practice in this style of

programming. Second, the computing curriculum leaves little, if any, room to add more courses, or more material into existing courses. Both cases, though, provide students with exposure to the material for limited times, and the students — in the time-honored tradition of college students — are likely to remember the material only so long as they need it to pass the class(es). So the focus should be on enabling the students to gain the knowledge, skills, and abilities necessary to make secure programming an integral part of their programming style.

We are developing a Secure Programming Clinic (SPC) to help students gain and practice the knowledge, skills and abilities (KSAs) needed to become a Secure Programmer. We intend to initially develop and test the Secure Programming Clinic as we investigate the efficacy of this instructional method for shaping students' defensive programming KSAs. Once we have an effective model, we plan to disseminate the clinic to other universities. A first essential step in our work is a Delphi study to identify the most critical knowledge in secure programming and create a concept map based on our findings. We intend to use this concept map to guide us in the development of the SPC, and to investigate students' developing mental models of secure programming.

#### **IV. Identifying Knowledge, Skills and Abilities**

##### ***A. The Delphi Study***

Given that the goal is to enable students to gain the knowledge, skills, and abilities necessary to make secure programming an integral part of their programming style, we have tried to identify the primary knowledge areas in secure programming. We used the Delphi Method to develop a graphical representation (concept map) of the “core” secure programming content. The Delphi Method is a structured technique that gathers input on the given topic from a panel of experts. The experts iterate their input to a given question(s) in two or more rounds. After each round, a facilitator provides a summary of the experts' inputs from the previous round as inputs for the experts to revise their earlier answers in light of the replies of other members of their panel. Our Delphi Method gathered information from ten experts over four rounds. A steering committee established a questionnaire containing seventeen (17) initial core characteristics related to C/C++ programming statements contributing to security relevant programming issues in the software development industry today (Figure 1). This questionnaire was distributed to the experts for comment. The facilitator received responses from five government experts, one industry, and four academic experts in assured software development. The respondents were asked to rate the set of characteristics according to five levels of importance (Very Important, Important, Somewhat Important, Not Important, and I disagree with this principle). The responses also encouraged additional contributions if the responders felt there were any core characteristics missing. The resultant set of thirty-one (31) core characteristics (Very Important, Important, and Somewhat Important ones) were consolidated over four (4) rounds of review. The importance levels roughly aligned with secure programming principles, concepts, and techniques, respectively.

**Secure Programming Clinic**

**Survey: Major Concepts in Secure Programming**

The purpose of this survey is to compile a list of the core concepts in secure programming based on feedback from experts like yourself. These concepts will be emphasized in the development of a secure programming clinic intended to support students in the development of secure programming skills. The primary programming languages for the clinic are C and C++.

**Procedure**

The following are a list of concepts that have been proposed as important by the team running the secure programming clinic.

A. For each one please indicate the importance of the concept

Concept	Not important	Somewhat important	Important	Very Important	I disagree with this principle
1. Assume that whatever can go wrong will					
2. If you don't generate it, don't trust it					
3. Hide details that users don't need to know about					
4. Assume any input is going to be malformed or not what you expect					
5. If it cannot happen, check for it. Someone may modify the program in such a way that it can happen ... or you may be wrong					
6. Define a list of acceptable characters and have the program ignore or discard any non-acceptable characters					
7. Do not use string functions that do not perform any bound checking					
8. Do not use input functions that cannot check the length of the input					
9. When the memory a pointer points to is freed, the pointer should be reset to NULL. Otherwise, these dangling pointers could cause writing to freed memory, and create a double free vulnerabilities					
10. Check parameters to ensure that all arguments are of the correct type and will not overflow any arrays					
11. Use data abstraction to enable the compiler to perform rigorous type checking and to enforce constraints on values and lengths					

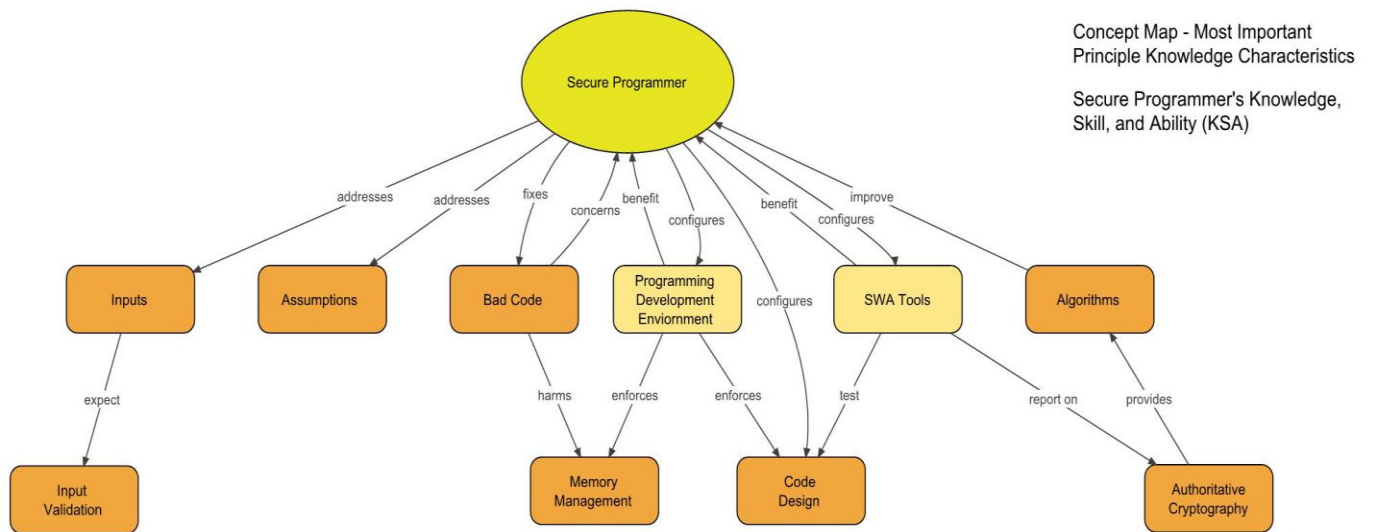
12. Avoid side effects in arguments to unsafe macros. If a developer is using a macro that uses its arguments more than once, then the developer must avoid passing any arguments with side effects to that macro					
13. Use parentheses around macro replacement lists. Otherwise operator precedence may cause the expression to be computed in unexpected ways					
14. C and C++ compilers generally do not check types rigorously. A developer can increase this level of checking by turning on compiler warnings, which will often catch more type errors than if they are not used					
15. Avoid calls to malloc() with the parameter (number of bytes to be allocated) set to 0. Either the function returns NULL, or it returns a pointer to space that cannot be used without overwriting unallocated memory					
16. Choose appropriate termination options					
17. Minimize the scope of variables and functions. This prevents many unexpected changes to the variables due to programming error					

B. Please list any additional concepts that you think are missing from this list

*Figure 1: Delphi Protocol with initial seventeen items*

## B. Findings

The Delphi Study resulted in a set of thirty-one core characteristics grouped as Very Important, Important, and Somewhat Important. Characteristics ranked as Not Important were discarded. These thirty-one core characteristics will be emphasized in the development of the SPC to support students in the development of secure programming skills. In order to understand the relationships of these core concepts, the steering committee developed a concept map in collaboration with some of the experts. The concept map takes the core characteristics and connects them hierarchically using a set of principles and relationships (Figure 2). This concept map was organized using Bloom's revised Taxonomy of Learning as a guide [KD02]. This revised taxonomy lays out four dimensions of knowledge ranging from the concrete to the abstract: factual, conceptual, procedural, and meta-cognitive. For our purpose, Factual Knowledge corresponds to learning basics; the program language instructions and their basic operation. Conceptual Knowledge is where a learner begins combining various basic elements with larger structure relationships, combining programming techniques with important security concepts. The final two dimensions - Procedural Knowledge and Metacognitive Knowledge - are achieved when the learner combines the basic elements, structure relationships, and can apply and combine the knowledge base to solve new problems based on appropriate contextual and conditional knowledge. This corresponds to combining programming techniques and security concepts with very important security principles to culminate in a learner that is a Secure Programmer.



A Concept Map provides a representation of critical areas of knowledge, skills, and abilities needed by a programmer in order to become a secure programmer. This view shows some of the most important principles (rounded rectangle shape) for a Secure Programmer (oval shape) to have learned and be able to apply to design, develop and deliver software programs with the appropriate level of trust and quality to achieve the stakeholder and use requirements of the system. The premise is that each area relates to at least one other area. The areas are noun or noun phrases and the relationships are connected with directional arrows that are labeled with relationships. They may be read as; <a noun> verb -> <another noun>. For example; a Secure Programmer fixes Bad Code and also, Bad Code concerns a Secure Programmer.

Figure 2: Secure Programmer Concept Map - Principles Only

The steering committee for the SPC recognizes the set of principles is not complete. For instance, a principle area on deployment environment embodies the knowledge level necessary to assess and address the impact of the deployment environment. This knowledge has a critical impact on design decisions about the program to be developed. Another missing principle area is that of risk analysis, which again contributes to software design, architecture and coding decision-making. However, the steering committee felt that the set of principles included in the concept map is sufficient to provide valuable instruction to the introductory secure programmer who is the primary target of the SPC. The steering committee recognized that in addition to mastering the knowledge of the material, it is also important to master the tools available. In our concept map we have two sets of tools. One is “Programming Development Environment” and one is “SWA (SoftWare Analysis) Tools”. Tools are invaluable and necessary to learn, and to learning, at all levels of secure programming from a novice to an expert, so they are included in the Concept Map.

The completed Secure Programmer Concept Map (Fig. 3) shows how the aggregation of secure programming characteristics, in concert with the skilled application of the set of programming tools, intend to provide a Secure Programmer practical KSAs. Fig 3 offers that the growth of knowledge in an individual begins with basic characteristics Somewhat Important which combine and contribute upwards in levels of knowledge and importance. Always the growth is ultimately contributing into Principle levels. It also demonstrates how some characteristics begin at higher levels, such as, the two Very Important characteristics 10 and 11. They are related to complex issues associated with the Principle of Authoritative Cryptography.

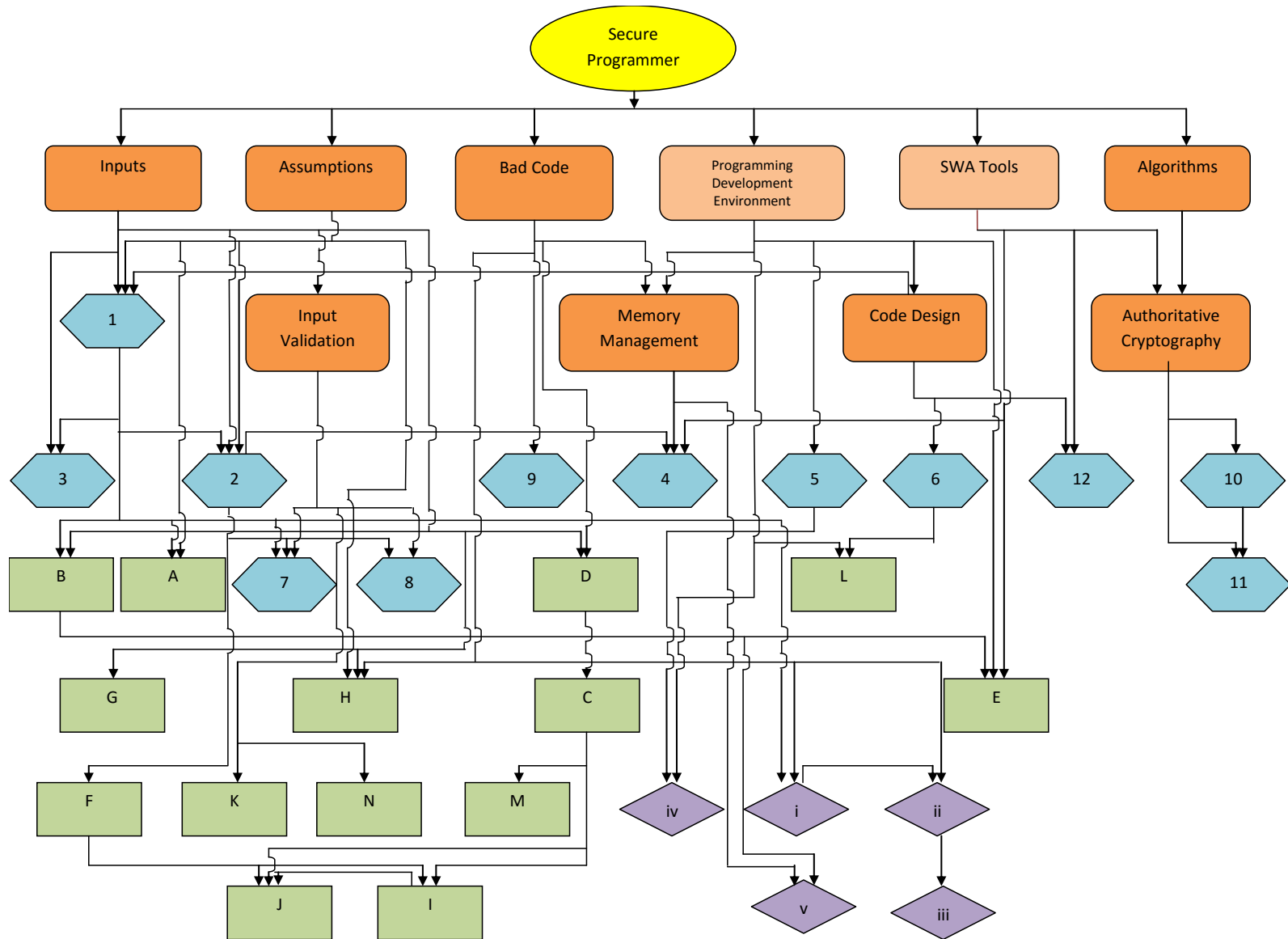





Figure 3: Secure Programmer Concept Map - Complete Set of Knowledge, Skills, and Abilities (KSAs)



- Very Important 
1. Assume whatever can go wrong will
  2. Assume any input is going to be malformed or not what you expect
  3. Do not make a security decision based on un-trusted inputs
  4. Check parameters to ensure that all arguments are of the correct type and will not overflow any arrays
  5. Use data abstraction to enable the compiler to perform rigorous type checking and to enforce constraints on values and lengths
  6. Understand the context in which the program will execute
  7. Validate your input stream to ensure that the commands invoked are expected and no other commands are injected
  8. When performing input validation take into account how programs invoked with those arguments could interpret them
  9. Avoid hard coded passwords and secrets in your program
  10. Use well known and accepted cryptographic algorithms and implementations of those algorithms. Don't use obsolete or deprecated cryptographic algorithms or create your own algorithms
  11. Use well known and accepted cryptographic random number generation. Don't use obsolete or deprecated cryptographic algorithms or create your own algorithms
  12. There are many tools to help you create a secure program, please take advantage of them

- Somewhat Important 
- i. Hide details that users don't need to know about
  - ii. Avoid side effects in arguments to unsafe macros. If a developer is using a macro that uses its arguments more than once, then the developer must avoid passing any arguments with side effects to that macro
  - iii. Use parentheses around macro replacement lists. Otherwise operator precedence may cause the expression to be computed in unexpected ways
  - iv. Minimize the scope of variables and functions. This prevents many unexpected changes to the variables due to programming error
  - v. When the memory a pointer points to is freed, the pointer should be reset to NULL. Otherwise, these dangling pointers could cause writing to freed memory, and create a double free vulnerabilities

- Important 
- A. If you have no reason to trust it, don't trust it. Take greater care with any input you have not generated
  - B. If it cannot happen, check for it. Someone may modify the program in such a way that it can happen ... or you may be wrong
  - C. Do not use input or constructor string functions that do not perform any bound checking
  - D. Do not use input or constructor functions that cannot check the length of the input
  - E. C and C++ compilers generally do not check types rigorously. A developer can increase this level of checking by turning on compiler warnings, which will often catch more type errors than if they are not used
  - F. Avoid calls to malloc() with the parameter (number of bytes to be allocated) set to 0. Either the function returns NULL, or it returns a pointer to space that cannot be used without overwriting unallocated memory
  - G. Control the input values when possible by limiting them to a finite set
  - H. Calling functions with null parameters for input should be checked for and defended against
  - I. Type conversion issues especially for cases that may result in integer wraparound and overflows
  - J. Rules for pointer arithmetic as vulnerabilities can arise when addition or size checks involve two pointer types
  - K. When performing input validation make sure that any validated path does not allow escaping from a restricted directory
  - L. Before creating a directory or file, make sure you have set the correct default permission specification
  - M. Be wary of off by one errors
  - N. When using format string functions, make sure that the format string can be authenticated/trusted

*Figure 3 (cont.): Secure Programmer Concept Map Key*

## V. Conclusion

The shortfalls in secure programming in many computer science programs may be addressed using a Secure Programming Clinic (SPC). A Delphi Study was undertaken to determine the core characteristics necessary to create secure programmer expertise. The Delphi study was an important step towards the design of the SPC program. The resultant Secure Programmer Concept Map will be used to design, teach, and evaluate the SPC.

## VI. References

[Bilt11] N. Bilton, "Bug Causes iPhone Alarm to Greet New Year with Silence," New York Times (Jan 2, 2011); available at <http://www.nytimes.com/2011/01/03/technology/03iphone.html>

[Bish02] M. Bishop, *Computer Security: Art and Science*, Addison-Wesley Professional, Boston, MA (Dec. 2002).

[CCK+11] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno, "Comprehensive Experimental Analyses of Automotive Attack Surfaces," Proceedings of the 20th USENIX Security Symposium (Aug. 2011).

[CERT99] *Buffer Overflows in SSH daemon and RSAREF2 Library*, CERT Advisory CA-1999- 15, CERT, Pittsburgh, PA, USA (Dec. 1999).

[Core99] CoreLabs Research, *Buffer Overflow in RSAREF2*, CoreLabs Advisory CORE-120199, CoreLabs Research (1999).

[DrDr80] S. Dreyfus and H. Dreyfus, (February 1980). *A Five-Stage Model of the Mental Activities Involved in Directed Skill Acquisition*. Washington, DC: Storming Media.

[EvRe10] K. Evans and F. Reeder, *A Human Capital Crisis in Cybersecurity*, Center for Strategic and International Studies, Washington, DC (2010).

[FDA10] Infusion Pump Improvement Initiative, Center for Devices and Radiological Health, Food and Drug Administration, Silver Spring, MD 20993 (Apr. 2010); available at <http://www.fda.gov/downloads/MedicalDevices/ProductsandMedicalProcedures//parGeneralHospitalDevicesandSupplies/InfusionPumps/UCM206189>.

[John10] R. Johnson, "More Details on Today's Outage" (Sep 2010); available at [http://www.facebook.com/note.php?note\\_id=431441338919&id=9445547199&ref=mf](http://www.facebook.com/note.php?note_id=431441338919&id=9445547199&ref=mf)

[KD02] Krathwohl, D. R. (2002), *A Revision of Bloom's Taxonomy: An Overview, Theory into Practice*, Vol. 41, Number 4, Autumn 2002, Copyright 2002, College of Education, the Ohio State University.

[NB12] K. Nance, B. Hay, and M. Bishop, "Secure coding education: Are we

Dark, M., Belcher, S., Ngambeki, I. and Bishop, M. (2015). Practice, Practice, Practice....Secure Programmer. *Proceeding of the 19th Colloquium for Information System Security Education*, Las Vegas, NV.

making progress?” Proceedings of the 16th Colloquium for Information Systems Security Education (June 2012).

[NRC00] National Research Council, *How People Learn—Brain, Mind, Experience, and School*, National Academy Press, Washington DC (2000).

[Palm04] C. Palmer, “Can We Win the Security Game?”, *IEEE Security and Privacy* **2**(1) pp. 10–12 (Jan. 2004).

[Schn14] B. Schneier, “Heartbleed,” *Schneier on Security* (Apr. 2014); available at <https://www.schneier.com/blog/archives/2014/04/heartbleed.html>

[Wein71] G. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, NY, USA (1971).

[Zett08] K. Zetter, “Serious Error in Diebold Voting Software Caused Lost Ballots in California County—Update,” *Wired* (Dec. 8, 2008); available at <http://www.wired.com/threatlevel/2008/12/unique-election>