Evaluating Secure Programming Knowledge

Matt Bishop[1], Jun Dai[2], Melissa Dark[3], Ida Ngambeki[4], Phillip Nico[5], and Minghua Zhu[6]

[1] University of California at Davis; *email*: `mabishop@ucdavis.edu`
[2] California State University at Sacramento; *email*: `jun.dai@csus.edu`
[3] Purdue University; *email*: `dark@purdue.edu`
[4] Purdue University; *email*: `ingambek@purdue.edu`
[5] California Polytechnic State University; *email*: `pnico@calpoly.edu`
[6] University of California at Davis; *email*: `mhzhu@ucdavis.edu`

Abstract. Secure programming is a widely used term for programming robustly. Applying the principles and methodologies of this style of programming would significantly improve the quality of software in use today. Teaching students how to program robustly, or securely, is a first step towards this goal. This paper presents a concept map for secure programming and then some questions used to evaluate students' knowledge of this subject. These questions have been given both before and after a term of programming, computer security, and other classes that cover this subject. In this paper, we discuss how the questions reveal the students' understanding of material in the concept map, and what erroneous ideas the questions reveal.

1 Introduction

In the United States, there has been considerable concern about the problem of poor software. One of the effects has been to examine how to teach students to program robustly.[7] There are two approaches commonly considered. The first, adding this material to classes, would require existing material to be dropped, and the instructors to understand and apply this style of programming to all class work. But in an algorithms class, for example, the students are assumed to know how to program, so the teachers and graders focus on whether the programs correctly implement the algorithms and meet the assignment requirements. Thus, whether the students write their programs securely is not considered, and — like any other aspect of practice — their skill atrophies (or is never acquired). The second, creating a separate class that covers the principles and practice of robust programming, adds a new class to an already crowded curriculum. The

[7] In this paper, we use the terms "secure" and "robust" synonymously. In practice, they are slightly different. "Secure" programming refers to a program that meets specific security requirements. "Robust" programming refers to programs that do not crash, and handle bad inputs in a reasonable way ("reasonable" being defined in the context of use). Nevertheless, people refer to "secure programming" when they mean "secure and robust programming", and we adopt this use.

class would need to be required to ensure students learned the material. But the students would need to practice it after the class, and this would require the co-operation of other instructors, resulting in problems similar to those identified earlier.

A third approach is to provide support for students through a mechanism other than a class [3]. Writing clinics in law schools and English programs do this. The clinics provide assistance on the mechanics of good writing: grammar, organization, and expression. They do not determine if the content is correct or meets the requirements of the assignment. With this clinic, the teachers can focus on the content, and leave the mechanics to others. Similarly, a "secure programming clinic" would assist students by showing them how to improve the robustness and security of their programs without determining if they satisfied the requirements of the assignment. Teaching robust programming techniques in the context of assignments where robustness is not the focus, motivates the students because they will see that programming robustly produces better overall results. For example, making a habit of checking library and system calls for error return values speeds development and produces more correct programs because programming mistakes are likely to be discovered quickly and remedied. While analysis tools can provide much of this information, the clinic focuses on student understanding. In addition to improving the students' programming, the knowledge obtained from the clinic enables the students to analyze the results produced by the tools, specifically to distinguish false positives from true positives.

A key part of starting such a clinic is to understand how students think about robust programming and to assess whether the clinic is having the desired effect on their understanding of secure programming. To do this we have designed an assessment consisting of a pre-test and a post-test administered to the students at the beginning and end of the term during which they use the clinic. These questions are the start of a concept inventory. In order to develop these questions, we started by developing a concept map of secure programming to make sure we were assessing a reasonable body of knowledge. We then developed questions based on the concepts represented in the map. This paper presents the concept map of secure programming and discusses several questions designed to test students' knowledge and misconceptions of secure programming based on the concept map. We explain how the questions relate to the concept map, and how well a set of 162 students performed when asked the questions.

2 Concept Inventories

Concept inventories are assessments designed to identify students' misconceptions; the questions, administration, scoring procedures, and interpretations are consistent and in adherence with a predetermined standard/protocol. They are not intended to be used to "grade" student learning, and therefore do not replace examinations, homework, discussions, and other methods used to measure student learning. Instead, the intent of concept inventories is diagnostic. Concept inventories are designed to measure core concepts of a topic and the extent to which students have achieved expert level thinking in a domain. The scores are used to tell us how many students do and do not understand a concept, and which conceptual picture they hold instead.

Concept inventories are useful for helping diagnose particular levels of students' conceptual understanding. Research on addressing misconceptions in science suggests that a new concept cannot be learned until the student is forced to confront the paradoxes, inconsistencies, and limitations of the mental model that already exists in the student's mind [7]. Students persist in erroneous beliefs, not intentionally, but because their erroneous beliefs seem plausible, useful, and accurate [11]. Concept inventories aim to identify these erroneous beliefs, so that once they are manifest, they can be addressed and corrected. The results from concept inventories are primarily intended to improve pedagogy, though the results can be used to help instructors make comparisons of teaching over time.

This method for assessing students' conceptual understanding was first developed 50 years ago, in the field of physics education to measure students' understanding of Newtonian forces [1, 9]. This initial concept inventory, the Force Concept Inventory, is still widely used today in physics and engineering. Since then several studies have validated the effectiveness of concept inventories for distinguishing between students who have understood concepts and students who merely memorized them [6]. Concept inventories have also been validated as effective in the evaluation of teaching methods [8, 10, 14]. Today concept inventories also exist in astronomy, chemistry, engineering design graphics, biology, thermodynamics, heat transfer, statics, statistics, electro-magnetism, circuit theory, genetics, nursing, and many other disciplines.

The concept inventory is based on the group of learning theories classified as conceptual change theories. Concepts are an abstract mental representation of a particular phenomenon. Conceptual learning therefore is the process of identifying and correctly categorizing concepts such that they can later be used to make predictions or decisions [4, 12]. Correct categorization involves making links to prior knowledge and so may require adjustment or correction of prior knowledge. Ausubel's assimilation theory contrasts rote learning (temporary acquisition of disorganized or poorly understood isolated or arbitrarily related concepts) with meaningful learning (long-term acquisition of organized, interrelated concepts into existing cognitive structures) [2]. In this theory, meaningful learning requires the connection of new knowledge to pre-existing understanding. This theory has been supported by subsequent research into student learning [4]. Another theoretical basis for concept inventories comes from the National Research Council publication, *Knowing What Students Know: The Science and Design of Educational Assessment* [13]. This study lays out an "assessment triangle" as the basis for assessment instruments. This triangle has three elements: cognition, a theory that describes how students learn in a particular content domain; observation, tasks that allow students to demonstrate their knowledge; and interpretation, a coherent method to make inferences about student knowledge based on observations from the assessment [14, 13]. Therefore, the assessment instrument must be designed to align the beliefs about how students learn in the content domain with the assessment tasks and interpretation of those tasks. Based on these theories, the development of the concept inventory begins with a clear understanding of the content domain, in this case secure programming, and an understanding of how students learn in this domain.

3 Concept Map and Inventory

Figure 1 shows the concept map developed for this project. Figure 2 describes each element of the concept map. This concept map depicts epistemologically important sets of concrete and abstract objects in secure programming. This concept map was developed using the input of subject matter experts. Development of the concept map is described in full in a related paper [5]. To date, this project has developed and tested 26 questions, based on the concept map, aimed at diagnosing students' conceptual misunderstandings in secure programming. The questions have been tested through the implementation of the Secure Programming Clinic at University of California Davis, California Polytechnic State University San Luis Obispo, and California State University Sacramento. Roughly half of the questions have been used and revised three times now, and half have been used and revised twice. The team continues to add additional questions with the goal of eventually covering all concepts depicted in the concept map. Examples of the questions and explanations of how distractors are used to target misconceptions are presented below.

4 Example Questions

The goal of the questions presented in this section is to determine how well the students understand the concepts underlying secure programming. The questions therefore have to have carefully designed distractors to ensure that the students who answer the questions correctly do so based on understanding the concept and not based on simply eliminating obviously wrong answers. The questions presented here were selected from a larger set of 27 questions. Each question has several distractors, the rationale for each is discussed, and the effectiveness of the distractors are shown as percentages of the 162 students who completed the assessment.

4.1 Handling User Input

This question deals with the important concept "If you have no reason to trust it, don't trust it. Take greater care with any input you have not generated." The goal of this question is to see if students know how to handle such input.

QUESTION: User input can be unpredictable. Which of the following is the best way to avoid problems processing that input?

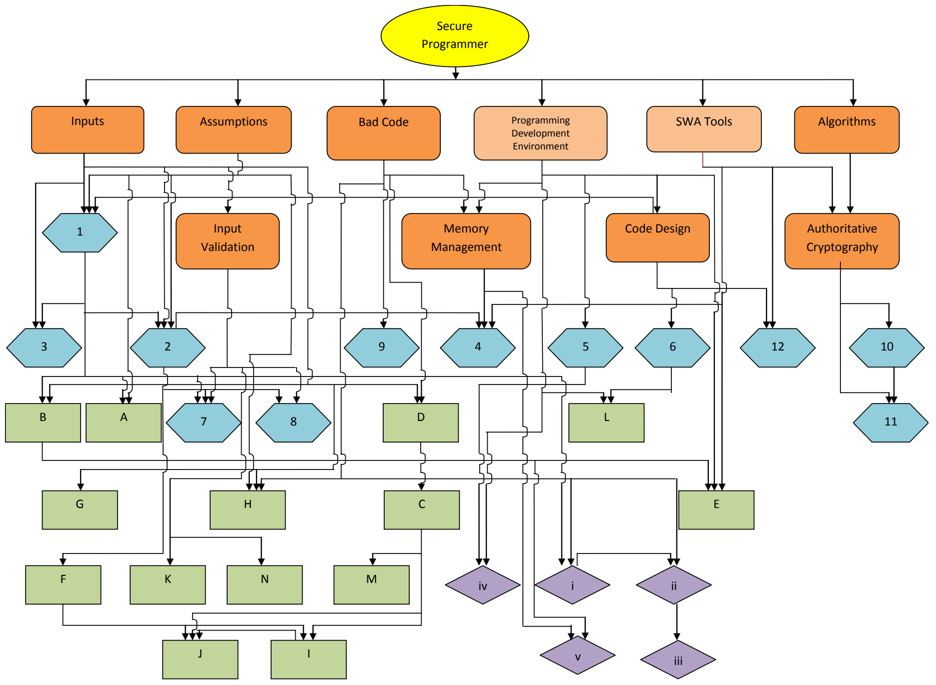

Fig. 1. The concept map. Figure 2 provides the captions for the content.

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 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. 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. Many tools 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, set the pointer to NULL. Otherwise, these dangling pointers could cause writing to freed memory, and create a double free vulnerability.

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

Fig. 2. The contents of the concept map.

a. Elevate privileges when processing user-provided input, to ensure the computation can be done.
b. Drop unnecessary privileges when processing user-provided input, to limit the effects of bad user input.
c. Keep privileges constant whenever possible, for more readable code that is easier to maintain without introducing error.
d. Assign elevated privileges to a new process or thread that reads the input and does the computation, so that any malicious side-effects do not affect the primary process or thread.
e. Keep privileges the same but constrain the process execution in a sandbox so that any malicious side-effects are contained.

The approach to answering this question lies in the effects of bad input, which could cause the program to act in unexpected ways. If the program has elevated privileges, this could breach security. The program may also serve as a vector for the attacker to inject malicious code, for example through a buffer overflow. The answers probe what the student is thinking about how to handle this situation.

Answer (a) focuses on the trade-off between security and getting the job done. Here, students selecting (a) are focused on the latter rather than the former. Answer (d) is similar, but here there is an element of isolation by handling the input in a separate thread. This presents an application of a secure programming concept; it is simply the wrong use of that concept. Answers (c) and (e) sound reasonable because no elevation of privileges is involved, but (c) focuses on simplicity and (e) the concept of isolation. The correct answer is (d), as it implements least privilege most effectively.

Most students (43%) got the right answer. The rest generally chose (d) and (e) (18% and 25%, respectively). A few (9%) chose (c), and only 5% chose (a).

The concept of isolation provides a good distractor. Students found that elevating privileges without isolation was obviously wrong, and keeping privileges constant without isolation was less obviously wrong, but generally considered wrong. So from these answers, an instructor can lead a discussion of the role of isolation in secure programming, and how to use it properly.

4.2 Indexing into an Array

Failing to check bounds and indices when manipulating arrays is a common problem. This question deals with the important concept "Check parameters to ensure that all arguments are of the correct type and will not overflow any arrays." It focuses on overflow in the negative direction (really, underflow) that is more subtle than overflow due to numbers that are too big.

QUESTION: Your program accepts parameters x, y, and z to calculate the position of an item in an array relative to the current item indexed by ptr.

```
101 newOffset = (x * colSize) + (y * rowSize) - z
102 ptr = ptr + newOffset
103 newObject = objectArray[ptr]
```

Which of the following is true?

a. I should check that the result in line 101 is not negative.
b. I should check that the result in line 101 is not null.
c. I should check that the result in line 102 is not negative.
d. I should check that the result in line 102 is not null.

This question examines the student's awareness of needing to locate the parameters to check and to validate the index used. Here, the parameters are implicitly of an integer type or a type that would be coerced into the integer type in arithmetic expressions.

Answers (a) and (b) both deal with validating something other than the index. It is tempting to assume checking newOffset is enough as that is derived directly from the parameters, but that is not the index; ptr is. They speak to locating the right variable to check.

Answers (b) and (d) both deal with the parameter type. The value "null" refers to a pointer that one cannot validly dereference (for example, the NULL pointer in C). As the result in line 102 is used as an index in line 103, it cannot be a pointer, and hence checking for "null" is not appropriate. Also, in many languages, "null" is a synonym for 0, which is a valid index.

When given to students, 38% got the right answer, (c). The rest of the students generally chose (a) (28%) followed by (b) and (d) (both at 17%). Thus, the three distractors effectively cover the most usual student misunderstandings, and resulting failures to index into an array properly in this code snippet.

4.3 Handling Missing Data

This question speaks to the important concept "If you have no reason to trust it, don't trust it. Take greater care with any input you have not generated." Here, the input from a list has some fields that are missing.

Question: You must read a list of user names and starting date: day, month, year. Then your program must sort them in ascending order to create a list of users by seniority. Some start dates are missing the day or month of the start date. This list-sorting function may be used elsewhere, or tweaked in the future. Which statement below is the most robust way to handle the missing data?

a. Initialize the variables for missing information with a random plausible value.
b. Leave the variables for missing information uninitialized.
c. Initialize the variables for missing information with 0.
d. Initialize the variables for missing information with the maximum plausible value.

This question examines how the student handles the missing data. In general, there are four ways to handle such data: ignore it, insert a value (random, 0, or a maximum value), give an error message and discard the rest of the record, or have the program stop. The answers focus on the first two behaviors. The key is that the caller of the sorting function must handle the missing values because the list-sorting function may be used elsewhere, or changed.

Answer (b) ignores the missing data. Answer (a) suggests using random values. The problem with this answer is that the data with the missing values will be randomly scattered throughout the data with all values present. Answers (c) and (d) cause the data with the missing fields to be grouped together, but as we do not know the minimum value for those fields, setting the missing values to 0 may put the data into the middle of all the data. Answer (d) puts all the data at one end of all the data, so it is the most robust.

When given to students, 22% got the right answer, (d). Of the other answers, 57% of the students chose (c). Fewer chose (a) and (b) (7% and 13%, respectively). Thus the incorrect answers provide good distractors to capture the student misconceptions for this question.[8]

4.4 Pointer Validation

This question tests common misconceptions about two important concepts, "Follow the rules for pointer arithmetic as vulnerabilities can arise when addition or size checks involve two pointer types" and "Be Wary of off by one errors." Problems with these concepts often occur together.

QUESTION: For a C program you must create an array of size integers. You write:

```
unsigned long *start, *end;
start = malloc(size * sizeof(unsigned long));
```

Assuming malloc succeeds, the correct value for end can be computed by:

a. end = start + size * sizeof(unsigned long);
b. end = start + size * (sizeof(unsigned long) − 1);
c. end = start + (size − 1) * sizeof(unsigned long);
d. end = start + size − 1;
e. end = start + sizeof(unsigned long) − 1;

This question tests the student's knowledge of the two concepts by asking the student to set a pointer to the last element of an array of unsigned longs. Answer (a) goes past the end of the array by the number of bytes in one unsigned long integer (usually 4 or 8 bytes). Answers (b) and (c) offer the option of subtracting 1 from either the size of the allocated space or the element size, thereby testing the student's understanding of pointers. Answer (d) captures those students who understand the off-by-one problem, but who do not realize that the buffer size and element size are distinct, and answer (e) omits the number of elements in the array. All these are common student errors.

For this question, 31% of students got the right answer, (c). Answers (b) and (e) distracted roughly 16% of students, while (d) attracted 20%. Even the least successful, (a), attracted 10% of the students. Thus the incorrect answers provide good distractors to capture the student misconceptions for this question.

[8] The numbers add up to 99%, not 100%, due to roundoff error.

4.5 Input Validation

This question speaks to the very important concept “Assume any input is going to be malformed or not what you expect.” The vehicle used is parameters that are pointers.

Question: You must write a function that stores an integer in the destination pointed to by value, and returns an integer indicating success or failure. You start with this function signature:

```
int getSeconds(int * secondsParameter)
```

Which of the following must you do before or instead of any of the others?

a. I must dereference the pointer to get the memory location.
b. I must find the value that the pointer refers to.
c. I must check that the pointer passed in does not already have a value.
d. I must check that the pointer passed in is not NULL.

Here, common misconceptions about pointers lead to confusion about what is and is not possible with input validation in C. The only thing that the programmer can do is to check that the pointer is not NULL, meaning the correct answer is (d). Answer (a) is based on the erroneous belief that dereferencing a pointer gets the address of the location rather than the value stored at the pointed-to location.. Answer (b) refers to the value at the location, but does not check that the pointer is not NULL. Answer (c) uses the common misconception among novice programmers that something special must be done with respect to initializing or not initializing a pointer before it can be used.

Most students (69%) got the correct answer. Distractors (a) and (c) were similar in effectiveness, capturing 10% and 13% of the students respectively. Answer (b) got 8% of the students. Hence most of the students understood the idea underlying the concept of input validation.

4.6 Use of Tools

This question deals with the important concept “There are many tools to help you create a secure program, please take advantage of them.” The goal of this question is to see if students know the differences between a file descriptor and a file name.

Question: Explain the choice of a file descriptor over the file name as the channel to securely access a file.

a. A file descriptor is a data structure that allows only me to use the file for as long as it is open, while the file name does not.
b. The file descriptor is an abstraction that makes for more understandable code.
c. The file descriptor is a pointer to the file that stays the same regardless of changes to the file name or location.
d. The file descriptor is a data structure that encapsulates the file name.

e. The file descriptor is a data structure that represents the validated file name.

The approach to answering this question lies in the differences between a file descriptor and a file name. A file descriptor designates an open file in a particular process, and a file name is a string that designates a file in a filesystem. One common problem in computer programming is the TOCTTOU (Time Of Check to Time Of Use) race condition. Using a file descriptor to a file allows one to check or change the status on the file without worrying that the file reference has been changed in the meantime.

Answer (a) is wrong because multiple file descriptors can refer to the same file. Answer (b) is not correct because "file descriptor" has nothing to do with making code more understandable. Answers (d) and (e) are wrong because the file descriptor is an integer that the kernel uses as an index into a table containing information about the file. The "file descriptor is a data structure" in answers (a), (d) and (e) is a good distractor. Thus, those selecting (a), (d) and (e) do not understand the underlying mechanism of a file descriptor. The correct answer is (c), because the file descriptor refers to the same file regardless of changes to the file name or location.

When given to students, 54% got the right answer. The rest generally chose (a), (d) and (e) (11%, 14% and 12%, respectively). A few (9%) chose (b). This suggests that an instructor should focus on the underlying mechanism of a file descriptor.

5 Psychometric Analysis

Once students have taken the test, the test question and its distractors are analyzed. We calculate item effect, which is a point biserial correlation coefficient with a possible range of -1.00 to 1.00. Item effect tells us which students with a high overall score also got a particular test question correct. A strong positive correlation suggests that students who get any one question correct also have a relatively high score on the overall exam. For a concept inventory where the main goal is to identify misconceptions, item effect is most useful for identifying questions that are not functioning at all, i.e., those that have a very low correlation or a negative correlation. Such a correlation would indicate that the distractors are so confusing even students who generally know the material are unable to answer correctly. Figure 3 below shows the item effect distribution for the 26 questions on the concept inventory.

We then combine the item effect with an analysis of the percent of correct responses for each item distractor. The table in Fig. 4 below shows the item effect and distractor analysis for a sample question (with content removed). The correct answer is (c). This test question is functioning fairly well; it is both discriminating knowers from non-knowers, and the distractors are not so easy that students can guess they are not the right answer.

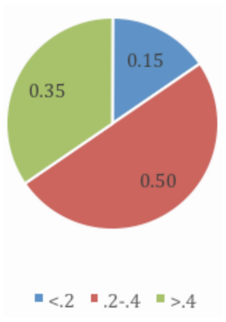

Fig. 3. Item effect distribution for the 26 questions on the concept inventory.

Test Item	**% correct response**
(a)	27.78%
(b)	17.28%
(c)	38.27%
(d)	16.67%
Item effect	0.363

Fig. 4. Example of distractor analysis and item effect for a sample question.

6 Conclusion

Three schools have used these questions in an evaluation of students in three different classes. The questions here were used to test students' understanding at the beginning of the classes. They could also be used at the end, and the difference will indicate how well the students have absorbed the material on secure programming. A second set of questions has been developed for testing at the end of the classes, using the same concept map and a similar method of evaluating the distractors.

In the future, we hope to involve more academic institutions in this program. The greater the diversity of schools, and hence students, involved in the evaluation of these questions, the more effective the distractors under a wide variety of circumstances. This will lead to clearer results in the evaluation of the secure programming clinics, the ultimate goal of these questions.

Acknowledgements. We Thank Stephen Belcher for his valuable insights and assistance with this work. This material is based upon work supported by the National Science Foundation under Grant No. DGE-1303211 to the University of California at Davis and Grant No. DGE-1303048 to Purdue University. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. Abou Halloun, I., Hestenes, D.: The initial knowledge state of college physics students. American Journal of Physics 53(11), 1043–1055 (1985)
2. Ausubel, D.P., Novak, J.D., Hanesian, H.: Education Psychology: A Cognitive View. Holt, Rinehart & Winston, New York, NY USA (1978)
3. Bishop, M., Orvis, B.J.: A clinic to teach good programming practices. In: Proceedings from the Tenth Colloquium on Information Systems Security Education. pp. 168–174 (Jun 2006)
4. Bransford, J.D., Brown, A.L., Cocking, R.R.: How People Learn: Brain, Mind, Experience, and School. National Academies Press, Washington, DC USA (2000)
5. Dark, M., Ngambeki, I., Bishop, M., Belcher, S.: Teach the hands, train the mind . . . a secure programming clinic! In: Proceedings of the 19th Colloquium for Information Systems Security Education. pp. 119–133 (Jun 2015)
6. D'Avanzo, C.: Biology concept inventories: Overview, status, and next steps. BioScience 58(11), 1079–1085 (Dec 2008)
7. Garvin-Doxas, K., Klymkowsky, M., Elrod, S.: Building, using, and maximizing the impact of concept inventories in the biological sciences: Report on a national science foundation–sponsored conference on the construction of concept inventories in the biological sciences. CBE Life Sciences Education 6(4), 277–282 (Winter 2007)
8. Hake, R.R.: Interactive engagement versus traditional methods: A six thousand student survey of mechanics test data for introductory physics courses. American Journal of Physics 66(1), 64–74 (Jan 1998)
9. Hestenes, D., Wells, M., Swackhamer, G.: Force concept inventory. The Physics Teacher 30(3), 159–166 (Mar 1992)
10. Laws, P., Sokoloff, D., Thornton, R.: Promoting active learning using the results of physics education results. UniServe Science News 13, 14–19 (Jul 1999)
11. Mayer, R.E.: Educational Psychology: A Cognitive Approach. Harper Collins, New York, NY USA (1987)
12. Özdemir, G., Clark, D.B.: An overview of conceptual change theories. Eurasia Journal of Mathematics, Science & Technology Education 3(4), 351–361 (2007)
13. Pellegrino, J.W., Chudowsky, N., Glaser, R.: Knowing What Students Know: The Science and Design of Educational Assessment. National Academies Press, Washington, DC USA (2001)
14. Streveler, R.A., Miller, R.L., Santiago-Román, A.I., Nelson, M.A., Geist, M.R., Olds, B.M.: Rigorous methodology for concept inventory development: Using the 'assessment triangle' to develop and test the thermal and transport science concept inventory (ttci). International Journal of Engineering Education 27(5), 968–984 (2011)