Towards Finding the Missing Pieces to Teach Secure Programming Skills to Students

Majed Almansoori
University of Wisconsin - Madison
malmansoori2@wisc.edu

Jessica Lam
University of California, San Diego
jplam@ucsd.edu

Elias Fang
University of California, San Diego
efang@ucsd.edu

Adalbert Gerald Soosai Raj
University of California, San Diego
gerald@eng.ucsd.edu

Rahul Chatterjee
University of Wisconsin - Madison
rahul.chatterjee@wisc.edu

ABSTRACT

Research efforts tried to expose students to security topics early in the undergraduate CS curriculum. However, such efforts are rarely adopted in practice and remain less effective when it comes to writing secure code. In our prior work [18], we identified key issues with the how students code and grouped them into six themes: (a) Knowledge of C, (b) Understanding compiler and OS messages, (c) Utilization of resources, (d) Knowledge of memory, (e) Awareness of unsafe functions, and (f) Understanding of security topics. In this work, we aim to understand students' knowledge about each theme and how that knowledge affects their secure coding practices. Thus, we propose a modified SOLO taxonomy for the latter five themes. We apply the taxonomy to the coding interview data of 21 students from two US R1 universities. Our results suggest that most students have limited knowledge of each theme. We also show that scoring low in these themes correlates with why students fail to write secure code and identify possible vulnerabilities.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; • **Security and privacy** → **Vulnerability management**.

KEYWORDS

SOLO taxonomy, Computer systems, Teaching security

ACM Reference Format:
Majed Almansoori, Jessica Lam, Elias Fang, Adalbert Gerald Soosai Raj, and Rahul Chatterjee. 2023. Towards Finding the Missing Pieces to Teach Secure Programming Skills to Students. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569730

1 INTRODUCTION

Our society, its infrastructure, and the economy are increasingly reliant on software. Therefore, it is important that developers who write and maintain such software are at least aware of basic computer security to ensure that these applications are secure and robust against malicious attacks. Unfortunately, developers lack basic security skills [14], and they routinely write vulnerable code [23].

Computer science students in the US can graduate without taking any security course [2, 22]. To combat the lack of security knowledge, prior studies have tried to introduce security topics in students' early computer science (CS) courses [15, 17, 21, 25], design security clinics to assess students' code [6–8], and develop tools to teach secure programming feedback [24, 28, 29]. Unfortunately, undergraduate CS students still lack basic security knowledge, such as safely reading a string from the standard input. While several prior works tried to propose a solution to this problem, we took a step back and tried to better understand the problem itself.

In our prior work [18], we found various knowledge areas that students lack but are prerequisites for writing secure code. We grouped our observations into six themes: (a) Knowledge of the programming language (C, in our case), (b) Understanding compiler and OS messages, (c) Utilization of resources, (d) Knowledge of memory, (e) Awareness of unsafe functions, and (f) Understanding of security topics. Although prior work provided what students lack, it did not delve into what level of knowledge of these skills is essential for writing secure code. Understanding this would help us prioritize the areas that we must teach to our students. Thus, we ask: *How could we categorize and evaluate students' understanding of prerequisite knowledge and skills needed to write secure code?*

In this paper, we dive deeper into the themes proposed in prior work. For this, we extend the SOLO taxonomy [12] to assess students' understanding of topics that we believe are necessary for writing and maintaining secure code. We added four levels of understanding according to SOLO taxonomy, defined them in the context of required skills to write secure code (the taxonomy can be found in the supplementary materials [3]) and used the taxonomy to explain students' security mistakes in the code.

We found that most students had a low-level understanding of the five themes, and that these students failed to write secure code, showing a correlation between these themes and secure programming. Only a few students showed high-level understanding of these themes and wrote secure code successfully; these students generally completed some security course in the past.

We believe that this work will help rethink the computer science undergraduate course curriculum to focus on the problems we highlight. Doing so will help students write secure and robust code.

Theme	Observation	#
Compiler/OS messages	- Ignores warnings - Does not understand compiler's message	4 8
Resources	- Skims page or documentation - Checks example uses of functions only	15 10
Memory	- Mentions undefined behavior - Fails or struggles to fix overflow	7 11
Unsafe functions	- Used at least one unsafe function - Used safer alternatives for security reasons	17 5
Security topics	- Writes insecure code - Suggests wrong mitigation to buffer overflow	14 11

Figure 1: The number of students (#) per observation. Check supplementary materials [3] for the full list of codes.

2 RELATED WORK

Teaching students security. Research on teaching security is growing, especially for students in their early careers. This is important because prior work [2] has shown that students may graduate from the top R1 universities in the US without completing any security course. We assume that it is likely the same case for other universities around the world. We believe that students must learn about some security topics in their required courses to avoid writing trivial insecure code. However, Taylor and Sakharkar [26] showed that many textbooks used by database courses in the top 50 U.S. universities do not discuss the security implications of SQL injection or ways to defend against the vulnerability. Almansoori et al. [2] examined the mid-level computer systems course taught at 16 U.S. universities, and showed that unsafe C library functions (e.g., `gets`, `strcpy`, etc.) were used in both student projects and instructor-provided code snippets. These unsafe functions can lead to many security vulnerabilities such as buffer overflow. It was also shown that the textbooks used for these computer systems courses often do not discuss security at all and sometimes even contained unsafe functions without providing safer alternatives [4].

Security clinics and interventions. Prior work [19] showed that it is possible to teach security topics to students as early as CS1. As a result, several modules were created to teach students secure coding habits in these introductory courses [16]. Bishop and Orvis [8] created a security clinic to teach students secure coding practices beyond their introductory courses, which was proven to be successful in increasing students' security awareness and reducing the security issues found in their assignments. However, it was noted that students often only employed these secure programming practices when it was required of them in the assignment, seeing it as a secondary rather than an essential practice.

Continued usage of unsafe functions. However, despite these interventions, students continue to use unsafe functions [2, 18]. Lack of knowledge of these security implications can make it difficult for students to recognize similar issues in code should they come across them in the future. This is important because it was found in prior work by Fischer et al. [13] that the Stack Overflow platform, which is relied on by many software developers, contains an alarming amount of unsafe code snippets. If students are not prepared or do not understand the security implications of the code they read and write, it is possible that they will continue to employ unsafe coding practices in the future. Therefore, it is crucial to understand why students still write insecure code despite all the prior work trying to instill secure coding practices.

SOLO Level	Definition
Pre-structural	Fails to understand or have wrong understanding
Uni-structural	Has simple understanding or knows a single aspect of the topic.
Multi-structural	Understands many aspects of the topic, but not necessarily the connection between these aspects.
Relational	Has deep understanding of different aspects and the relation between them.
Extended Abstract	Has the ability to extend their deep knowledge to new applications and concepts.

Figure 2: Definition of levels used in SOLO taxonomy [5].

3 METHODS

We are using the data collected in our prior work [18], where we conducted a study in two R1 universities, UCSD and UW-Madison, and recruited 21 participants. We interviewed students who are at least 18 years old, familiar with C, and have taken Computer Systems or an equivalent course. Only 4 students completed security courses. Participants were interviewed over Zoom while they work on a coding test (the coding survey can be found in the supplementary materials [3]). Our work was approved by IRB as discussed in our prior work [18]. We came up with six themes that reoccurred during our interviews in our prior work [18]. These themes are the 1) knowledge of C programming, 2) understanding compiler and OS messages, 3) utilization of resources, 4) knowledge of memory, 5) awareness of unsafe functions, and 6) understanding of security topics. In this work, we are focusing only on the last five themes as knowing C programming is a prerequisite for our study. We coded our observations under each of the five themes for each student. The finalized codes are listed in Figure 1. Looking ahead, we will use these observations to develop a systematic taxonomy of different mistakes that lead to insecure code writing.

Extending SOLO taxonomy. Biggs and Collis introduced the structure of observed learning outcomes taxonomy, known as the SOLO taxonomy [5], to assess the depth of students' understanding of different concepts and topics. Since its introduction, SOLO taxonomy has been widely used by educators and researchers for assessing learning outcomes [10–12]. To our knowledge, few prior studies have tried to extend the SOLO taxonomy to assess students understanding of security. Although studies tried to apply Bloom's taxonomy [9] (e.g. [27]) to assess computer security education programs created by security professionals in the lens of an education theory, the only work that we could find that used SOLO taxonomy in the context of security and cryptography was by Patterson et al. [20]. The work, however, focuses on cryptography only, and does not consider understanding secure coding skills.

We extend the SOLO taxonomy in the context of writing secure code. Based on the definitions of different levels in SOLO taxonomy (Figure 2) and the observations we made from the interview data (Figure 1), we design a novel assessment procedure for students. We did not consider the extended abstract level because we are not expecting students to be able to apply their security knowledge

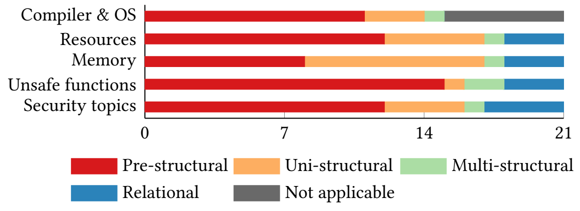

Figure 3: Distribution of students based on the SOLO levels.

and skills to new fields and concepts; we believe this is the job of security experts. Moreover, our coding questions were designed to only assess students' basic understanding of security and applying them while reading and writing codes. Thus, our data is insufficient to evaluate whether students have extended abstract knowledge.

We iterated over the structure and the definitions of the taxonomy multiple times. We made the observations based on students' thinking-out-loud and their approach to interacting with the code editor, compiler, and their search behavior, to classify the students' understanding of security topics based on our taxonomy. The final version of the taxonomy can be found in the supplementary materials [3]. Using the taxonomy, two researchers coded the student interviewees independently. Then, the coders met to resolve any disagreement, and if a resolution could not be met, the research team met and collectively decided on a code through discussions.

4 RESULTS

We assess students' answers to the coding survey using our extended SOLO taxonomy (found in the supplementary materials [3]).

4.1 Understanding Compiler and OS Messages

Students were allowed to compile their code during the interview study; 16 students used their preferred IDEs, and 15 students got compiler (or OS) messages. Thus, they will be our focus in this subsection. As shown in Figure 3, most students had low-level understanding of compiler and OS messages (11 pre-structural and 3 uni-structural). Only one student showed multi-structural knowledge, but no student had relational understanding in this theme.

Ignoring Messages. Developers need to care about compilers and OS messages since they can indicate the need for security considerations. We observed that 11 students completely ignored all messages prompted by the compiler and proceeded without reading them. These students have written vulnerable code in our interview as well. For example, students got a warning about `gets` whenever they tried to use it. In one case, a student got the warning stating that *"The '`gets`' function is dangerous and should not be used."* for Q1. Despite the warning, the student executed the code with input "Hello World" and got "stack smashing detected" along with an incorrect output. Yet, the student did not pay attention to the output nor the error. Unfortunately, the student chose `gets` as one of their final answers and said that despite noticing the warnings: *"I think it works"*.

We noticed that students ignore warnings in general, since warnings do not mean the code is broken completely. However, students also ignored error messages, such as "core dumped", if they believed the code works as intended. In fact, similar to the aforementioned case, many other students got the "stack smashing detected" error when testing their code. Only one student was curious about why the error appears when they input a long string to `scanf` in Q4.

"I am not understanding messages." Although some students noticed compiler messages, many of them did not quite understand how to address them. Students were allowed to search online during the interview, but multiple students did not investigate these messages further. In one case, a student tried to compile the code we provided for Q3 using Visual Studio and got an error that says: *"`strcpy`: This function or variable may be unsafe. Consider using `strcpy_s` instead."* Although the student noticed the error, they thought that the library containing the function is not included, which was not the case. The student believed the error could be fixed by simply importing it; however, the student saw that the issue persisted. As a result, they said: *"I don't know"* and proceeded without trying to understand the message.

Some students who were prompted with "stack smashing detected" message noticed the error but did not look it up, nor showed any understanding of it. For example, one student got the "stack smashing detected" error once in Q4 and ignored it, then the student got the same error again in Q5. This time, the student noticed it but was not sure what it meant. The student was curious about the error, and after experimenting with the code, they realized that the print statement after the offending line is executed despite hitting the "stack smashing detected" error, and wondered whether the code is executed line by line: *"It's interesting how the stack smash detected comes in if I comment out the print statement. When I uncomment it, the print happens before the stack smashing detected. Does it just print everything first and then give me all the list of the errors or is it like line by line?"* Although the student noticed the error, they concluded that the correct answer is the one causing the error. The student stated that they recall learning about stack smashing before but do not remember anything about it.

In many cases, students cared only about the final result and whether the code works. For example, one student was attempting to find the issue with the code in Q4 and tried a long input. When the student got the stack smashing error, they said: *"It worked, but it [compiler] says stack smashing detected. Maybe this compiler is more advanced than the one people normally use, but it actually worked."* The student was willing to proceed with their attempts to find out how the code may break. However, the student decided to search the error online after we asked about it and told them that they may use online resources. After searching it online and reading, the student learned about the issue, which helped them recognize the error in the next question: *"So I would think that it could be one of the same errors that we had before"*.

4.2 Utilization of Resources

We do not expect students to know all functions and compiler messages; however, it is important to effectively utilize different resources (e.g., library documentation) in order to write efficient and secure code. Thus, students were allowed to use any resources.

Looking up Functions. Most students looked up at least one function, regardless of whether they already knew it or not. Interestingly, many students who checked documentation just skimmed them and never attempted to read them carefully. These students usually check the types of parameters passed to the function and proceed with using it. Similarly, some students checked only examples provided and attempted using functions in a similar fashion as these examples by simply changing the parameters, which might work in some cases, but fail in other. Indeed, we observed some students who failed to use some of the functions correctly because they did not pay attention to the documentation. For example, one student who stated that they know the difference between `fgets` and `gets`, skimmed the documentation of both functions, then chose them as correct answers for Q1, although `gets` is incorrect and does not print the desired output. Similarly, when attempting to answer Q2, a student found `strcat` in Stack Overflow after searching up "c string concatenation" and tried to use the function immediately without trying to learn how to use it correctly:

```
char* concatenated str = strcat(buf, argv[1]);
```

In another example, a student searched up `strcat` and checked the examples in *cplusplus.com*. The student decided to use it and then noticed `strcpy` in the same example, so they switched to it without checking its documentation. Not only the student failed to answer Q2, they thought that `strcpy` concatenates strings and did not even pay attention to the example provided in the website.

When a student was attempting to explain the issue in the snippet in Q3, they checked up the Linux manual page for `strcpy` (man7.com). The manual has an emphasized warning in different colors, stating "*Beware of buffer overruns!*". The page also has an entire section about bugs which discusses the issue with a small destination buffer. Although the student checked the manual multiple times, they were just skimming and ignored most of the text. The student then stated that they did not know why the values stored in the second array changed and proceeded to the next question.

Understanding Resources. In addition to library documentation, there are other resources that students encounter during coding such as Stack Overflow. We noticed that students often skimmed these resources and did not pay attention to the information presented in these results. Students mostly visited Stack Overflow when trying to find answers to their questions. When checking Stack Overflow answers, students usually search for a short and straightforward answer. We observed that if the answer seemed long or contained many lines of code, the student is more likely to skip it and leave the page. Moreover, when browsing answers, students would look for any function that might seem relevant and try to use it without reading the actual answer and try to understand it. Unfortunately, many of these students do not only skim search results but also copy answers and code without understanding them. One student searched up "concatenate c string" and copied the answer directly from Stack Overflow and modified it slightly. The answer contained many lines of code and a long explanation, but the student skimmed the top of the answer then copied a line of code without trying to understand the context. The student's final answer was:

```
char buffer[1024];
strcat(strcat(buffer,buf), argv[1]);
```

Although the answer works, if the input is not large, the student unnecessarily used `strcat` twice and created a new destination buffer other than the one provided in the code snippet. Both lines were copied from Stack Overflow. Prior work [13] has shown that Stack Overflow answers often contain vulnerable code snippet, and in this case, out of context copy-paste of code made it vulnerable.

4.3 Knowledge of Memory

Students are expected to learn and understand how memory works after taking a computer systems course. However, only three students showed substantial knowledge regarding memory (Figure 3).

Limitations of "Undefined Behavior." Many students limit their understanding of memory-related issues by thinking of them as "undefined/unexpected behavior". Prior work showed that lectures and textbooks for computer systems usually refer to buffer overflow and other memory-related issues as undefined behavior [4], and this could be one reason that students use this term frequently. Students who used this term showed limited understanding of buffer overflow and memory in general. For example, when attempting Q3, some students answered that we are copying 30 characters into a 15 character long buffer, which causes undefined behavior. A student said: *"The [buffer] size needs to be large enough when using* `strcpy`*. Otherwise, it results in undefined behavior."* Many of these students did not know the consequences of this undefined behavior.

A few students showed a better understanding of memory and buffer overflow despite mentioning undefined behavior. One student, while answering Q5, stated that the longest option would cause unexpected behavior because the input would overwrite values and mess up with other memory addresses. However, the student stated that long inputs would not crash the program and selected the wrong answer. Generally, some students do not know whether undefined behaviors could break the program or not. In an extreme case, a student believed that undefined behavior is not necessarily bad and stated: *"Like it [undefined behavior] is not always bad; it is just we don't know what is going to happen."* Referring to memory-related errors as undefined behaviors limits the understanding of their consequences. We argue that students should understand the actual effects of these errors on memory since they might introduce security vulnerabilities in programs.

Fixing and Avoiding Buffer Overflow. Many students thought that buffer overflow would be completely avoided by just increasing the size of the destination array. They usually suggested making the buffer large enough (e.g., 1KB) to handle all possible inputs. Nonetheless, students failed to realize that a malicious user can input strings longer than the expected input length. A student stated when answering Q4: *"There's going to be some point where there's an upper bound to someone's name, so you could probably conclude that there's a certain size that you can do and you'll be fine. You could just use some really big numbers or google whoever has the longest name and make it [destination buffer] slightly bigger. I don't know about any other ways."*

Many of the students who suggested increasing the buffer size also suggested dynamically allocating the array instead of static allocation. While this solution might seem plausible, it actually does not solve buffer overflow in most cases. If the user input is passed as a command-line argument, then it is possible to dynamically allocate an array with the size of `len(argv[1])`, avoiding an overflow in this case. However, if user input is passed to `stdin`, then a buffer should be already allocated before reading the input, meaning that the allocated buffer might be smaller.

A few students were categorized as having a multi-structural or relational knowledge of memory because they successfully prevented buffer overflow. These students paid attention to the buffer size and stopped reading the user input before overflowing the array. We asked these students whether increasing the buffer size was sufficient for solving the issue or not; they stated that you could never know the length of user input.

4.4 Awareness of Unsafe Functions

Most students showed no concern when using unsafe functions. 15 students had pre-structural knowledge about unsafe functions, and 1 student had uni-structural knowledge. Only a few students showed a better understanding of these functions: 2 students possessed multi-structural knowledge and 3 had relational knowledge.

Unfamiliarity with Unsafe Functions. When choosing a function that could accomplish a certain task, such as string concatenation or reading a standard input, students often chose the first function that came up in their Google search, as discussed earlier. This, combined with limited knowledge of unsafe functions, caused students to use unsafe functions in their code.

Unfortunately, students have shown very limited, or in many cases, no knowledge about unsafe functions. We hoped that students would know about `gets` at least and avoid using it, since prior work has shown that it is warned against in some textbooks and lectures [2, 4]. However, many students used `gets`. For example, a student suggested in their answer replacing `scanf` with `gets` to fix the issue in Q4: *"Using a different buffer-reading function (e.g.,* `gets`*) would fix it."* – (Written answer for Q4). The student did not just suggest using `gets`, but also did not figure out that `scanf` could be overflown in the context of Q4, which is concerning. In general, students did not just use `gets`, but also used `strcat`, `strcpy`, and `scanf`, suggesting that most students have no prior knowledge about unsafe functions and their flaws.

What is more concerning is that some students also used safer alternatives in their code incorrectly and without checking for buffer bounds. For example, some students used `fgets` instead of `gets`. However, their choice was not based on security considerations. Rather, they found the function online and decided to use it. In one instance, a student wrote the following code for Q1:

```
if (NULL == fgets (name, 1024, stdin)) { return 1; }
```

The student imitated the example provided in the documentation of `fgets` but failed to notice that the documentation limited the number of characters read to 60, which is the size of the buffer. Instead, the student changed 60 to 1024 without any considerations of the actual buffer size which was 12. Such a mistake when using safer alternatives would make them as bad as unsafe functions.

Ignoring Flaws of Function. While we encourage teaching about unsafe functions, learning about them is not enough to avoid writing insecure code. Surprisingly, some students still used some unsafe functions despite knowing about them. For example, one student, who has taken a security course, used `gets(name)` for Q1 at first. Then the student remembered that the survey was about secure coding habits, so they changed their answer to `fgets(name, 12, stdin)`. When using `fgets`, the student paid attention to the buffer size and avoided buffer overflow. In Q2, the student, however, used `strcpy` –which is vulnerable– along with `strncat` as follows:

```
strcpy(buf, strncat(buf, argv[1], 14));
```

Although this code snippet is safe in the given context, it uses `strcpy` in an unsafe way. The student paid attention only to the security implications of `strcat` versus `strncat`, and not to `strcpy`. While some unsafe functions can be used safely by ensuring that the source string is constant, we believe that avoiding these functions is a better practice for early stage programmers.

Another student who has also taken a security course showed a similar behavior: they used `gets(name)` for Q1. Then, while answering Q2, the student remembered about buffer overflow and said: *"I forgot to check for buffer overflow [in Q1]. I realized I didn't check the length of the input."* The student did not mention explicitly that `gets` is unsafe; however, they stated that they would use `getc` instead and read 10 characters only. In both cases, despite having a security background about unsafe functions and buffer overflow, both students were not coding with a security mindset.

4.5 Understanding of Security Topics

Many topics in the computer systems course, such as the ones related to memory, are closely related to security. As shown in Figure 3, most students have either pre-structural or uni-structural knowledge of security topics (a total of 12 and 4, respectively).

"I can just write code from scratch." Writing secure code requires following best practices. One of the best coding practices is to use safe and standard library functions. We however found students sometime try to write their own code instead of using standard library functions, and in the process inadvertantly introduce bugs and vulnerabilities (that could have been avoided by using the library functions). For example, in Q2, some students implemented string concatenation from scratch instead of using library functions. While the code might work correctly, the possibility of writing buggy code is higher than when using existing library functions. One example code from a student:

```
char buf[28] = "Hello ";
int offset = 6;
for (int i = 0; i < 22; i++) {
    buf[i + offset] = argv[1][i];
    if (i == sizeof(argv[1])) { break; }
}
```

The student tested this code snippet to ensure it works; however, the code breaks out of the loop whenever `i == sizeof(argv[1])`, and since `sizeof(argv[1])` is always 4, the code will break out of the loop before filling up the buffer.

In another example, a student also attempted to implement a string concatenation algorithm from scratch. However, in this case, the student made a small mistake:

```
char buf[28] = "Hello ";
int len = strlen(argv[1]);
for (int i = 0; i < len && i < 23; i++) {
    buf[i + 6] = argv[1][i];
}
```

When investigating the code closely, one can notice that the loop continues until `i < 23` or in other words, `i = 22`. Once the loop reaches the end, the program will be executing the following line:
`buf[28] = argv[1][22];`

Thus, this will allow writing one byte to the array out of bound, which can lead to buffer overflow vulnerabilities [1].

User Inputs Might be Long. When handling buffers, students mostly suggested increasing the buffer size or allocating the buffer dynamically. However, some students exhibited behavior that should be avoided at all costs. In one example, a student proposed asking the user to input a string shorter than the size of the buffer: *"This [buffer overflow] can be fixed by either asking [the user] for shorter input or reallocating space in name [destination buffer] to support the length of the input."* The student, in this example, assumed that the user is honest. Mistrusting the user is a desirable habit that we want to instill in students. In fact, we observed this behavior in a few students. One student suggested using `fgets` instead of `scanf` for Q4 to avoid buffer overflow. We asked the student whether increasing the size of the buffer would solve the issue; the student replied: *"We never know since the user might be a mischievous user."* The student also added that long inputs can be used for hacking.

5 DISCUSSION

We show there is a correlation between students' knowledge of the five essential skills and their secure C coding practices. To inculcate a security mindset, we may need to redesign some of our beginner-level CS courses.

Need to Focus on Improving Overall Coding Practices and Skills of Students. Prior work mainly focused on evaluating and improving students' secure programming knowledge and skills. However, as observed in Section 4, the real problem lies in students' lack of more fundamental knowledge and skills, such as paying attention to compiler and OS messages and carefully reading documentation. Students lacking knowledge about the five essential coding practices are prone to write insecure code.

Therefore, introductory CS courses (CS1 and CS2) should focus on instilling students to pay attention to compiler and OS messages and learn to utilize online and offline resources. Having a relational level of understanding of these two themes would help students improve their other required skills. Moreover, utilizing resources and paying attention to messages are among the most critical skills students would use after graduating and joining the industry. Moreover, courses that teach C or C++ languages should spend more time discussing unsafe functions and their security alternatives. Finally, computer systems courses should emphasize basic security mistakes and memory-related issues. We cannot expect students to acquire excellent secure programming skills and knowledge without understanding the basic details of process memory layout; students must grasp topics related to memory before learning about security. Ideally, students should be formally introduced to computer security through a dedicated course; however, given the lack of such required courses [2], students should be exposed to safe and unsafe functions in computer systems courses.

Ecological validity. We interviewed only 21 students across two R1 universities in the US. Although this is small, we believe that our results can be generalized to students (and developers) in real-life scenarios. First, we prepared an interviewing environment that would reduce and possibly eliminate any stress caused by actual interviews or exams. Students were allowed to spend as much as 90 minutes solving the survey questionnaire; we believe this provided students with more time than actually needed to finish the interview. Second, students could access all necessary resources to solve the questions on hand, unlike typical coding exams at universities. Third, we did not explicitly ask students to use compilers or online resources to evaluate their knowledge of each of the five themes. Finally, we interviewed students from two different universities, yet, we observed similar behaviors at both locations, suggesting that students at other universities might have similar coding skills.

Limitations. Our survey questionnaire was originally designed to evaluate the understanding of buffer overflow, which made it easy for some students to predict some of the questions. Redesigning the survey to avoid predicting questions might help find additional observations. Also, it is possible that students did not take the survey seriously because it was not graded nor was part of coursework. Finally, while we tried to reduce sources of stress as much as possible, we cannot say for sure that students were not stressed at all, especially since interviews were conducted during the beginning of the pandemic. Conducting the same study in a lab setting might yield different observations or results.

Future work. Our analysis shows that students lack some essential skills and knowledge necessary for writing secure C code. In order to teach students secure coding practices, it is important to identify and understand all the pieces that contribute to secure programming. Therefore, research should focus on finding the root cause for why students write insecure codes before trying to solve the issue. Moreover, our extended SOLO taxonomy [3] can be used to design surveys and further understand how each of the five themes contributes to secure programming skills.

6 CONCLUSION

We conducted coding interviews with 21 students from two R1 universities in the US to evaluate students' secure programming practices. Our prior work [18] showed that students lack skills and knowledge in five key themes contributing to security skills. To better understand how each theme contributes to secure coding skills, we designed a modified SOLO taxonomy for each of the five themes and used it to evaluate students. Our assessment showed that most students have a rudimentary knowledge and understanding of the essential skills (themes) needed for secure programming. We also found that students who had pre-structural or uni-structural knowledge of these themes generally failed to write secure code and avoid vulnerabilities, implying that there is a strong correlation between secure programming and these essential skills.

ACKNOWLEDGMENTS

We thank all the students who participated in our study. We also thank the anonymous reviewers for their feedback on our work. This work was supported in part by NSF Award 2044473. 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] CWE-193: Off-by-one error. https://cwe.mitre.org/data/definitions/193.html.

[2] Majed Almansoori, Jessica Lam, Elias Fang, Kieran Mulligan, Adalbert Gerald Soosai Raj, and Rahul Chatterjee. How Secure are our Computer Systems Courses? In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 271–281, 2020.

[3] Majed Almansoori, Jessica Lam, Elias Fang, Adalbert Gerald Soosai Raj, and Rahul Chatterjee. Supplementary Materials for Towards Finding the Missing Pieces to Teach Secure Programming Skills to Students (Published in SIGCSE TS 2023). https://www.majedalmansoori.com/papers/SIGCSE23_Supplementary_Materials.pdf.

[4] Majed Almansoori, Jessica Lam, Elias Fang, Adalbert Gerald Soosai Raj, and Rahul Chatterjee. Textbook Underflow: Insufficient Security Discussions in Textbooks Used for Computer Systems Courses. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 1212–1218, 2021.

[5] John B Biggs and Kevin F Collis. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press, 1982.

[6] Matt Bishop. A Clinic for "Secure" Programming. *IEEE Security and Privacy*, 8(2):54–56, 2010.

[7] Matt Bishop, Melissa Dark, Lynn Futcher, Johan Van Niekerk, Ida Ngambeki, Somdutta Bose, and Minghua Zhu. Learning Principles and The Secure Programming Clinic. In *IFIP World Conference on Information Security Education*, pages 16–29. Springer, 2019.

[8] Matt Bishop and BJ Orvis. A Clinic to Teach Good Programming Practices. In *Proceedings of the 10th Colloquium for Information Systems Security Education*, pages 168–1174, 2006.

[9] Benjamin S Bloom et al. Taxonomy of Educational Objectives. Vol. 1: Cognitive Domain. *New York: McKay*, 20(24):1, 1956.

[10] Gillian M Boulton-Lewis. The SOLO Taxonomy as a Means of Shaping and Assessing Learning in Higher Education. *Higher Education Research and Development*, 14(2):143–154, 1995.

[11] Claus Brabrand and Bettina Dahl. Using the SOLO Taxonomy to Analyze Competence Progression of University Science Curricula. *Higher Education*, 58(4):531–549, 2009.

[12] Charles C Chan, MS Tsui, Mandy YC Chan, and Joe H Hong. Applying the Structure of The Observed Learning Outcomes (SOLO) Taxonomy on Student's Learning Outcomes: An Empirical Study. *Assessment & Evaluation in Higher Education*, 27(6):511–527, 2002.

[13] Felix Fischer, Konstantin Bottinger, Huang Xiao, Christian Stranksy, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *2017 IEEE Symposium on Security and Privacy*, pages 121–136, 2017.

[14] Dave Gruber. Modern Application Development Security, 2020.

[15] Sara Hooshangi, Richard Weiss, and Justin Cappos. Can the Security Mindset Make Students Better Testers? In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 404–409, 2015.

[16] Cynthia E Irvine and Shiu-Kai Chin. Integrating Security into the Curriculum. *Computer*, 31(12):25–30, 1998.

[17] Siddharth Kaza and Blair Taylor. Introducing Secure Coding in Undergraduate (CS0, CS1, and CS2) and High School (AP Computer Science A) Programming Courses. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 1050–1050, 2018.

[18] Jessica Lam, Elias Fang, Majed Almansoori, Rahul Chatterjee, and Adalbert Gerald Soosai Raj. Identifying Gaps in the Secure Programming Knowledge and Skills of Students. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, pages 703–709, 2022.

[19] Kara Nance. Teach Them When They Aren't Looking: Introducing Security in CS1. *IEEE Security and Privacy*, 7(5):53–55, 2009.

[20] Blain Patterson. Analyzing Student Understanding of Cryptography Using the SOLO Taxonomy. *Cryptologia*, pages 1–11, 2020.

[21] Ambareen Siraj, Nigamanth Sridhar, John A Drew Hamilton Jr, Latifur Khan, Siddharth Kaza, Maanak Gupta, and Sudip Mittal. Is there a Security Mindset and Can it be Taught? In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, pages 335–336, 2021.

[22] Ludwig Slusky and Parviz Partow-Navid. Students Information Security Practices and Awareness. *Journal of Information Privacy and Security*, 8(4):3–26, 2012.

[23] C Symantec. Internet security threat report: Volume 24. *Symantee Enterprise Security*, 2019.

[24] Madiha Tabassum, Stacey Watson, Bill Chu, and Heather Richter Lipford. Evaluating Two Methods for Integrating Secure Programming Education. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 390–395, 2018.

[25] Blair Taylor and Siddharth Kaza. Security Injections@Towson: Integrating Secure Coding into Introductory Computer Science Courses. *ACM Transactions on Computing Education (TOCE)*, 16(4):1–20, 2016.

[26] Cynthia Taylor and Saheel Sakharkar. '); DROP TABLE textbooks;– An Argument for SQL Injection Coverage in Database Textbooks. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 191–197, 2019.

[27] Johan Van Niekerk and Rossouw Von Solms. Using Bloom's Taxonomy for Information Security Education. In *Information Assurance and Security Education and Training*, pages 280–287. Springer, 2013.

[28] Michael Whitney, Heather Richter Lipford, Bill Chu, and Jun Zhu. Embedding Secure Coding Instruction into the IDE: A Field Study in an Advanced CS Course. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 60–65, 2015.

[29] Jun Zhu, Heather Richter Lipford, and Bill Chu. Interactive Support for Secure Programming Education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 687–692, 2013.