Identifying Gaps in the Secure Programming Knowledge and Skills of Students

Jessica Lam\textsuperscript{1}, Elias Fang\textsuperscript{1}, Majed Almansoori\textsuperscript{2}
Rahul Chatterjee\textsuperscript{2}, Adalbert Gerald Soosai Raj\textsuperscript{1}
\textsuperscript{1} University of California, San Diego, \textsuperscript{2} University of Wisconsin - Madison
\{jplam,elias\}@ucsd.edu, \{malmansoori2, rahul.chatterjee\}@wisc.edu, gerald@eng.ucsd.edu

\textbf{ABSTRACT}

Often, security topics are only taught in advanced computer science (CS) courses. However, most US R1 universities do not require students to take these courses to complete an undergraduate CS degree. As a result, students can graduate without learning about computer security and secure programming practices. To gauge students’ knowledge and skills of secure programming, we conducted a coding interview with 21 students from two R1 universities in the United States. All the students in our study had at least taken Computer Systems or an equivalent course. We then analyzed the students’ approach to safe programming practices, such as avoiding unsafe functions like gets and strcpy, and basic security knowledge, such as writing code that assumes user inputs can be malicious. Our results suggest that students lack the key fundamental skills to write secure programs. For example, students rarely pay attention to details, such as compiler warnings, and often do not read programming language documentation with care. Moreover, some students’ understanding of memory layout is cursory, which is crucial for writing secure programs. We also found that some students are struggling with even the basics of C programming, even though it is the main language taught in Computer Systems courses.

\textbf{CCS CONCEPTS}

\begin{itemize}
  \item General and reference → Evaluation:
  \item Social and professional topics → Computer science education:
  \item Security and privacy → Vulnerability management.
\end{itemize}

\textbf{KEYWORDS}
Computer security education; Computer systems; Unsafe functions; Buffer overflow; Security vulnerabilities; C and C++

1 INTRODUCTION

Since security courses are often not required for a Computer Science degree, students can graduate without ever taking a security course. Furthermore, as prior work [2, 3] has shown, basic security topics are often not covered in the required courses, and students can graduate without learning the most basic — yet important — secure programming practices. This is particularly concerning given the fact that students will join the software developer workforce who design and build the digital systems that our modern society and critical national infrastructures rely on.

Prior studies [6, 11, 12, 18–20] have attempted to remedy this by developing and designing tools, modules, and interventions to address the lack of security topics in computer science education. Despite several prior works on improving students’ security knowledge, studies [2, 21] have shown that students regularly write insecure code in their assignments. An obvious remaining question is why students fail to write secure programs. Do they have all the required knowledge and skills to understand the potential security issues of the code they write?

Specifically, in this work, we aim to understand the following research question:

\textbf{RQ:} What are some common issues students face while writing secure programs and identifying, understanding, or fixing insecure code?

To answer this question, we conducted coding interviews with students who have at least taken Computer Systems or an equivalent course in two R1 universities (the University of California, San Diego (UCSD) and the University of Wisconsin-Madison (UW-Madison)) in the US. We developed the interview questionnaire to gauge students’ basic security knowledge and their approach to programming in terms of security. By approaching this as a qualitative study, we were able to focus on students’ behaviors and their actions to solve the problems they were given. Through this study, we hope to understand the issues that students face while working with code that may contain security issues and find the root causes for difficulties students face with secure programming.

In total, we interviewed 21 students over Zoom, with the participants sharing their screens during the interview. We analyzed the nearly 20-hour-long interviews and used coding and thematic analysis to analyze our data systematically. Overall, we found that many students lack the basic building blocks needed to write secure code: Students struggled with reading and understanding compiler warnings and language documentation, knowledge of memory, basic C programming, and knowledge of unsafe functions and their safer alternatives.
2 RELATED WORK

With the increased importance of security, it is necessary that all computer science students learn about the topic. However, security is often only taught in advanced courses, and many top universities in the US do not require students to take such security courses to graduate [2]. As a result, students can graduate without any knowledge of security and secure programming practices.

Since we cannot just rely on security experts to fix all vulnerabilities – including simple ones – in our code, we must train software engineers to develop secure coding practices by teaching security early in their CS curriculum. Prior work [16] has shown that we can introduce security concepts to students as early as CS1 to increase their security awareness from the beginning of their studies. There have also been modules [6, 11, 12, 18–20] designed to teach students secure coding habits in introductory CS courses (CS0, CS1, and CS2). These modules have been shown to effectively improve students’ security awareness and ability to apply security knowledge.

Bishop [5] explored the concept of a security clinic to teach secure coding habits beyond introductory courses. It was noted that students often only followed robust programming practices when required of them in a coding assignment and did not consider it as an essential practice. This study showed that implementing the clinic on a computer security course could help increase students’ security awareness and reduce security issues in their assignments.

Although there have been several studies on improving students’ security knowledge, students are still exposed to unsafe programming practices and thus continue to use them. For example, Taylor et al. [21] analyzed seven database textbooks used by the top 50 US CS programs and showed that many do not thoroughly discuss SQL injection, a common database exploit. They also found that these textbooks often do not teach ways to prevent SQL injection, such as writing parameterized queries, and that two of the seven textbooks even gave examples of code that was SQL injectable.

Recent studies [2, 3] have examined the mid-level Computer Systems course, which many universities teach. In this course, students are introduced to the C (or C++) language and some Assembly languages such as x86. Almansoori et al. [2] collected projects written by students in the top 20 CS programs in the US for their Computer Systems course and showed that students used many unsafe functions that can lead to security issues such as buffer overflow. Their work also showed that lectures did not warn about unsafe functions and even provided code snippets and project skeleton code containing some of those unsafe functions.

Moreover, Almansoori et al. [3] analyzed textbooks used in the Computer Systems course of the top 30 CS programs in the US and showed that most textbooks did not discuss security nor warn about using unsafe functions. Unfortunately, these textbooks also have vulnerable code snippets, possibly hindering students from writing secure code.

3 METHODOLOGY

In this section, we explain how we designed our survey, recruited participants, conducted interviews, and analyzed the collected data.

Designing the survey. We designed the survey to understand students’ approach to C programming questions with respect to security. We decided to mainly write questions that would evaluate students’ understanding of memory and stack-based buffer overflow exploits. With this in mind, we defined important behaviors to use when assessing students’ secure programming knowledge and skills; this list is shown in Figure 1. Based on these behaviors, we designed a set of questions and improved them iteratively, ending up with a set of five questions, as shown in Figure 1.

Prerequisites. We ensured that all students had already completed a Computer Systems course and learned C or C++ before taking the survey, as it was a prerequisite for students to participate. We also allowed participants who have not taken any security courses since this helped us understand whether security is taught well, if taught at all, in Computer Systems courses.

Interview questions. The first question evaluated a student’s knowledge of reading from standard input. We wanted to see if students could do so safely by giving students two function options: gets and fgets. We hoped that students would choose fgets to avoid running into buffer overflow by limiting the number of characters read. This also allowed us to observe whether or not students knew that gets is a deprecated and unsafe function that should be avoided while programming.

The second question tested students’ knowledge of the unsafe function strcat. Students were asked to fill in code concatenating user input passed in through the command line. For this question, we observed whether students would avoid the unsafe function strcat since the user can input any number of characters. Again, we looked for the use of safer alternatives, like sprintf.

The third question asks students to identify the error in a code snippet that contains the unsafe function strcat. The student must explain why the program changed the source array passed into strcpy and not the destination array, which requires knowledge of the stack and buffer overflow.

The fourth question asks students what potential issues a code snippet could have using scanf. Our goal is to test if the student would realize that the user could provide an input that is too long...
We also realized that the solution to the third question was easily
we wanted to recruit students who had completed Computer Sys-
was length 8 so the example input and output we provided often
Thus, we updated the sample code snippet and made the question
vulnerable to buffer overflow and understand which inputs could
fixing this error, we helped avoid any confusion or other external
The final question tests the ability to find out if a function is
Our survey also includes pre- and post-survey questions asking
about the interviewees’ year in school, the Computer Systems or
related courses that they have taken, their familiarity with C, C++,
Java, Python, and Assembly, their demographic information, and
whether or not they have had industry experience. The survey
questions can be found here https://bit.ly/2VTRqdc.
and thus cause a buffer overflow. We also asked for a potential fix,
for example, using fgets instead of adding a width specifier.
The final question tests the ability to find out if a function is
safe to use or not by generalizing prior knowledge about buffer
overflow. In this case, students needed to realize that sprintf is
vulnerable to buffer overflow and understand which inputs could
then cause the program to crash.
Our survey also includes pre- and post-survey questions asking
about the interviewees’ year in school, the Computer Systems or
related courses that they have taken, their familiarity with C, C++,
Java, Python, and Assembly, their demographic information, and
whether or not they have had industry experience. The survey
questions can be found here https://bit.ly/2VTRqdc.

### Improving the survey.
We conducted two pilot interviews at UCSD and realized that there were typos and mistakes in the first
draft of the survey. In the first question, the provided answer option
with the use of fgets contained a buffer of size 8 instead of 7. By
fixing this error, we helped avoid any confusion or other external
factors that could affect how the students answer that problem.
We also realized that the solution to the third question was easily
searchable on Stack Overflow, so we edited the question to be
unique and different. This removed the possibility of students just
copying and pasting answers from the Internet.
After conducting the UCSD interviews, we noticed that question
1 was not as helpful in answering our research question as we
originally expected. In the first iterations, the name character array
was length 8 so the example input and output we provided often
hinted to students that they needed to limit user input somehow.
Furthermore, because the first iteration listed answer options, we
could not observe other ways students could approach the question.
Thus, we updated the sample code snippet and made the question
more open-ended while conducting the UCSD interviews.

### Participant recruitment.
Students are typically introduced to
process memory and C/C++ in the Computer Systems course. Thus,
we wanted to recruit students who had completed Computer Sys-
tems and self-evaluated that they know how to code in C/C++.

We created a recruitment form that asked how often students
code in C and whether they have taken Computer Systems or an
equivalent course. We only recruited students who knew C, were
18 or older, and consented to be recorded. As an incentive, all
participants received a $15 dollar Amazon gift card. As our study
involved recruiting and interviewing students, we received IRB
approval from both universities to conduct our research.

We recruited 11 students from UCSD by distributing the recruit-
ment form in Discord servers of 13 student organizations and in
four Facebook groups. After we interviewed the students from
UCSD, we recruited 10 students from UW-Madison by emailing the
computer science undergraduate mailing list. In total, between two
universities, we interviewed a total of 21 students. Among these 21,
8 were female, and the remaining were male. Most of the students
had just completed their Junior year. The demographics of our par-
ticipants can be seen in Figure 2. Out of all of the participants, only
3 students had taken at least one security course, and one student
learned about security outside of school out of their own interest.

### Interviews.
For each interview, we ensured that the student consented to be recorded. We asked participants to follow a think-aloud
protocol to help us understand students’ thought processes during
the interview. On average, interviews lasted about 30-75 minutes.

For UCSD, interviews were conducted by two interviewers; one
focused on giving the student directions and answering the stu-
dent’s questions about the survey, while the other focused on taking
notes on the student’s survey responses and behaviors. Both inter-
viewers asked clarification questions about the student’s approach
and process. We also often prompted students to think aloud so that
we could follow their thought process more easily and efficiently.
For UW-Madison, we had only a single interviewer since we noticed
that having a note-taker was not necessary because the interviews
were video recorded. Otherwise, the protocol was the same.

During the interview, the students were allowed to use search
engines like Google, look up programming documentation, and
use their preferred IDEs to compile and test code. The goal was
to provide students with a typical coding environment as in the
real world so that we could observe how students would normally
behave while solving programming questions related to security.
We also refrained from answering student questions other than
clarification to help them understand the problems better to avoid
biasing our results.

Although our consent form mentioned that the interview would
contain security-related questions, we did not explicitly state during
the interview that we would assess students on their secure coding
practices. In doing so, we could more accurately observe whether
students would apply their security knowledge in typical coding
environments that are not necessarily security-focused.

After interviewing 10 students at UCSD and 11 students at UW-
Madison, we decided against interviewing additional students as we
did not observe any new behaviors among students after a certain
number of interviews.

### Data analysis.
After conducting all interviews, the interviewers reviewed each recording and transcript, taking notes on specific im-
portant behaviors. For example, how students initially approached

---

1IRB numbers for UCSD and UW-Madison are 201933SX and 2020-1574 respectively.
problems, what resources they used to solve problems, and whether or not the solutions they came up with were considered “secure”. Upon reviewing these notes as a team, we noticed several common themes present among students when they were solving each question. We then created multiple codes that we could associate with each theme to more clearly see the strengths and issues that students exhibited and faced during the interview and to help identify the main areas of concern regarding secure programming. The themes that emerged from these interviews are presented in Section 4.

4 RESULTS

Through the interview and coding process, we found that many students lacked the necessary fundamental knowledge in several areas, which ultimately hindered their ability to write secure code. On the other hand, we also observed several students who performed very well on certain questions, and thus we were able to take note of the kinds of useful prior knowledge or skills they utilized in these instances. Through the themes that emerged from the interview observations, we decided on 6 main areas of knowledge that we consider helpful prerequisites for learning how to code securely.

4.1 Understanding compiler messages

Compilers print useful messages, especially warnings and errors, which can help produce more secure code. However, we found that several students regularly ignored important compiler warnings that were pointing to security issues with the functions they were using. The primary reason for ignoring them is that the written code would still execute and output relevant results before failing. An example of this is when S9 (student 9) at UCSD tested the code they wrote for Q1 (question 1), which used the C library function gets. Their compiler gave warnings that stated gets is deprecated and unsafe. Still, the student continued to use the function in their answer anyway, stating, “We get a bunch of warnings about how it’s unsafe ... and how it’s been deprecated ... but I think it works.”

We also observed that students do not read compiler errors carefully or understand them properly. For example, while executing Q4, some students received a “stack smashing detected” exception due to inputting a long string into a small buffer. This exception is added to notify users about potential buffer overflow attacks. However, we found only 3 students were familiar with or had even heard about the stack smashing exception (S1 at UCSD and S6, S8 at UW-Madison). Many students either did not encounter the specific error or did not know how to address it. Some students believed it was an issue with their computer or compiler, and others just ignored the error and moved on. Only 5 students (S1, S9 at UCSD, and S1, S4, S7 at UW-Madison) who did not know about the error actually tried to look it up on the Web. In general, we found students who did well in identifying, understanding, and fixing the buggy code snippets also paid close attention to the warning and exception messages.

4.2 Utilization of resources

Students were allowed to use any online or locally-installed C compilers to test their code during the interview. We also allowed them to search up anything they would like in order to simulate a realistic software developer workflow. We observed how students use the available resources (function manuals and the internet) to understand and solve the programming problems.

We saw that at least 4 students (S1, S6 at UCSD and S1, S2 at UW-Madison) copied answers from online forums (most commonly Stack Overflow) or arbitrary examples on documentation sites. Moreover, students would often copy the first suggested answer or example code without reading the post in detail or other comments. This was worrisome since Stack Overflow often contains insecure code snippets [8]. It was also concerning that many students only skim through documentation and go straight to the examples section. As such, students sometimes missed how to correctly use the function and noticeably ignored important warnings about the security issues with certain functions.

For example, when addressing Q2, students searched up ways to concatenate strings in C. For many, top search results would reveal the C library function strcat, and they would simply use this function in their answer. As a result, only a much smaller subset of students who either read other Stack Overflow answers or read the documentation in-depth learned that, while strcat works, it is much safer to use its alternative strncat. The difference in the way students referenced online materials affected how securely they answered this specific question.

In an extreme case, S6 at UCSD accessed strcat documentation and saw an example on the same page using the function strcpy. Seeing the example, the student ended up answering using strcpy, assuming that it behaves similarly to strcat, stating that they were “using the strcpy function and [had] the buffer as the destination [with argv[1]] to be added on to the buffer”.

Something we were not expecting but were very excited to see was that some students took the initiative to explore a topic brought up during the interview and took the time to learn about it to better understand the question and how to answer it. For example, S10 at UW-Madison came across a Stack Overflow answer mentioning “buffer overflow” while trying to understand how to answer Q3 of the interview. As a result, the student searched up buffer overflow to better understand it.

4.3 Knowledge of memory

Since our interview questions focused on the buffer overflow vulnerability, understanding how processes use computer memory would be essential for answering the survey questions. However, many students displayed little knowledge or had misunderstandings about memory and memory layout.

For example, in Q3, many students realized that the issue was due to copying a string into a smaller destination buffer. Still, they could not recall the memory layout to explain the error further. In Q3, two character arrays were defined and are stored directly one after another in the stack. Therefore, copying larger content into the latter array overwrites the former character array. As a result, they did not understand why the buffer overflow caused this exact behavior and either skipped the question, gave vague reasoning, or cited that it was because of “undefined/unexpected behavior”. According to S1 at UW-Madison, “the size” of the destination “needs to be large enough when using strcpy. Otherwise, it results in undefined behavior”. Another student, S3 at UW-Madison, said, “something bad happens” if the buffer overflows.
Similarly, S7 at UCSD could not recall whether arrays are stored in the stack or the heap, and forgot the order in which arrays are stored in the stack. Since the student was given access to the Internet, they were able to find these answers eventually, but this helped show the lack of understanding they originally had about memory and memory layout. Another student, S2 at UW-Madison, also had confusion about how data are stored in memory. Although both buffers are stored in the stack, the student said, “question1 is allocated on stack memory, and question2 is allocated on heap”.

Other students have shown minimal to no understanding of memory errors. For example, S2 at UW-Madison said that a “Memory leak” caused the output of the code snippet provided in Q3. Moreover, the student stated that “the first 15 characters in question1 are now referencing memory that doesn’t contain any character values.” Other students have also discussed that dangling pointers could cause the issue.

4.4 Knowledge of C Programming Language

Although we required interviewees to be familiar with C, we found that many students struggled to recognize several standard C library functions and displayed several misunderstandings about the C programming language.

First, several standard C library functions appear throughout the survey and/or were used by students in the interview. These include printf, scanf, gets, fgets, strcpy, and strcat. Although many of them are popular functions, we noticed that even if some students recalled these functions, many did not remember the details and usage of a majority of them. Note that sprintf appears in the survey, but we do not expect students to have encountered this function before.

For example, when S8 at UW-Madison attempted to use the function strcat in Q2 of the survey, they misunderstood how it works and tried to use it in the following way: strcpy(buf, strcat(buf, argv[1], 14)). While strcat concatenates the source string to the destination string, the student believed that it returns a string of the destination and source strings concatenated together as a new string. Thus, the student used strcpy in an attempt to assign the “output” of strcat to buf.

Other students had more general misunderstandings of C. For example, S4 at UCSD could not recall if C has bounds checking, which is an important reason why much of C code is insecure to buffer overflow. In another case, some students (S6 at UCSD and S2, S4 at UW-Madison) had difficulty with Q3 and misunderstood that it was an issue with pointers being somehow rearranged when accessed.

4.5 Knowledge of unsafe functions

We separated this from the “Knowledge of C programming Language” category because, while some students may know or recognize some C library functions, they may not realize that the function is unsafe or has safer alternatives. This is important because understanding how to use the safer alternative of a function can be a simpler and quicker way to fix unsafe C code.

Arguably, the function gets is the most well-known unsafe library function [1, 15] as gets has been deprecated and also because it was mentioned as unsafe in most textbooks used by Introductory Computer Systems courses [3]. However, we found that roughly half of the students interviewed (S1, S2, S3, S4, S6, S7, S8, S9, S11 at UCSD and S4, S5 at UW-Madison) displayed that they either did not know gets at all or recognized it but still did not know of its security implications. As a result, a few of those students continued to use it in their code. Note that several students did not encounter gets at all during the interview, and so we could not assess whether they considered the function as unsafe or not.

On the other hand, several students either had prior security knowledge or had taken a security course and recalled the safer alternatives to some of the library functions tested (i.e., strncpy is a safer alternative to strcpy). Some students didn’t know that some of these functions were unsafe but learned about them and their safer alternatives when searching them up during the interview.

4.6 Understanding of general security topics

We did not expect the students who participated in this survey to have a huge security background. However, while students may not completely understand these security topics, some displayed a level of understanding of the implications and consequences of errors/vulnerabilities in the code in terms of security. Thus, we came up with this category to gauge how well students understood the code they read and wrote in terms of security.

In response to Q1, Q2, and Q4, several students trusted user input and believed that, by simply increasing the buffer size or by prompting the user to input less than n characters, they could solve this issue where the user may input something longer than the allotted buffer size. For example, S1 at UW-Madison explained, “I would change the buffer size to be a larger character string, maybe like 256 or 1024”. In another case, S11 at UCSD mentioned that the issue could be fixed by “either asking for shorter input or reallocating space in name to support the length of the input”. It is evident through these responses that these students were not considering this question with security in mind (as an attacker is not the average user and would thus still be able to exploit this program even after these proposed fixes).

However, even if students had prior knowledge of security, they did not always begin the interview with security in mind. For example, a student S9 at UW-Madison originally used the unsafe function gets in Q1 but, after submitting and upon approaching question 2, they realized that they “forgot to check for buffer overflow” in the previous question. When prompted, this student was able to articulate a more secure way to answer the first question, which showed that they did, in fact, understand how to code securely. However, coding with security in mind seemed to be more of an added afterthought in this case rather than the primary practice.

5 DISCUSSION

Through the interviews and coding process, we found six main areas of concern that hindered students’ understanding of secure programming. In this section, we interpret the results, discuss them in relation to prior studies, and provide recommendations to improve the status quo of security education. We also discuss the limitations of our work and possible directions for future work.

Interpretation of results. Currently, much of security education research focuses on students’ knowledge of security and on finding
ways to teach security through interventions like security clinics or learning modules on security [6, 11, 12]. However, our results show that perhaps we have failed to see the root causes of the problem – many students struggle with secure programming because they lack some fundamental knowledge/skills in programming.

Although students are not required to be fully proficient in all six categories listed above (see Section 4), we found that an understanding or lack thereof in any or all of these areas was important to a student’s overall understanding of secure coding. These six main categories thus also help pinpoint areas that the security education research community can focus on improving and therefore better prepare students to learn secure coding practices.

**Alternative interpretation.** Most students did not do well on our 5-question programming quiz, and this could be interpreted that the questions we asked were too difficult and tricky for the students. However, we restricted our questions to basic programming knowledge, such as copying a string, reading input, etc. Another limitation could be that we did not explicitly mention that we were testing their security skills and practices during the interview (Though the careful reading of our consent form will reveal the study intention, which few students read carefully).

**Relationship with prior studies.** The “understanding compiler messages” and “utilization of resources” categories highlighted that many students lacked or struggled with these important skills (that are usually not explicitly taught in a CS course). This is congruent with prior research on the complexity of compiler errors and warnings and the difficulties novice programmers face when addressing them [7, 17]. As such, there has been much work addressing ways to teach these skills as well as finding interventions to help students digest complex compiler messages [4, 9, 10] and more easily seek answers online [13, 14]. In the “knowledge of unsafe functions” category, we showed that many students did not recognize unsafe functions and the safer alternatives of many C library functions tested in the interview. However, this was expected as it was shown in prior works that Introductory Computer Systems courses (in which many students learn how to program in C) do not warn their students of these security issues [2], and textbooks used by these courses also do not always mention the safer alternatives of many C library functions [3].

**Recommendations** We realize that several of these categories, such as “understanding compiler messages” and “utilization of resources”, are not often taught in traditional computer science courses and thus may also be a contributing factor as to why students struggled in these aspects. However, we found these skills important for students to better understand the code they encounter and guide students toward more robust and secure coding practices. Thus, we propose that these important skills should receive more emphasis in the classroom.

We also found that students struggled with the C programming language. This may be because students often start with learning high-level programming languages such as Java and only receive introductions to C later in their university curriculum (for example, in an Introductory Computer Systems course). The introduction to the C programming language is often quick, brief, and taught alongside difficult concepts such as memory, bitwise operations, etc. This could explain why students struggled to recall many C library functions and concepts.

**Limitations and threats to validity** There were only 21 total interviews conducted over 2 different US universities. As such, our participant pool only represents a smaller subset of computer science students attending US universities. In order to gain a clearer picture and solidify our data, we would need to repeat these interviews with a more significant number of students across a wider variety of US universities. Our results were also collected through an interview process. Thus, with the added pressure of a perceived assessment, the collected results may not wholly reflect how students would normally approach such questions. The questions students were asked to answer were also on toy code snippets, which have little purpose outside their use in the interview, so students may not have responded to them as seriously as opposed to if they had to address more critical code, for example, in their workplace.

**Future work.** Currently, there is a gap in security education research that seeks to pinpoint these areas of concern and propose interventions for teaching or supplementing these knowledge areas. If we want to achieve our goal of teaching students secure coding practices, we should focus on finding these ways to bridge the gaps we found in prerequisite knowledge and skills. In doing so, we can better prepare students to learn secure coding practices and also use these teaching methods to supplement other ways of teaching security, such as the module-based approach.

**6 CONCLUSION**

We interviewed 21 students across two US universities to understand how students approach coding to identify the barriers they may face when attempting to understand or fix unsafe code and write secure programs. As a result, we found six main knowledge areas we consider essential foundations and prerequisites that can help students better employ secure coding practices. Students who excelled in the interview tended to display knowledge/skills in several of these six areas. On the other hand, we found that many students whose lack of knowledge/skills in one or more of these areas ultimately hindered their ability with secure programming. Through this work, we pinpoint the lack of understanding in these important prerequisite knowledge areas as root causes of the difficulty students faced when attempting to identify and fix insecure code. Beyond that, even for students who do have sufficient knowledge in these areas, it is important that they keep them in mind when reading and writing code. We hope these results bring awareness to the security education research community to focus on addressing ways to better teach, supplement, and demonstrate the importance of these knowledge areas to bridge the gap students are facing when employing secure coding practices.

**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


