How Secure are our Computer Systems Courses?

Majed Almansoori[1], Jessica Lam[2], Elias Fang[2], Kieran Mulligan[1]
Adalbert Gerald Soosai Raj[2], Rahul Chatterjee[1]

[1] University of Wisconsin - Madison [2] University of California, San Diego

ABSTRACT

Introductory computer systems courses teach students how a single program is executed inside a computer, providing them with their first exposure to the logical internals of computing systems. This is one of the first introductory courses where students can learn about security and the need for robust coding. However, currently, these courses are taught with a focus on functionality and efficiency only, ignoring security almost entirely.

In this paper, we provide a basic security analysis of computer systems courses from 16 of the top 20 CS undergraduate programs at R1 universities in the US. We collected more than 760 thousand lines of C/C++ code written by 253 students and used by instructors in lectures and for assignments. We found students frequently use unsafe functions such as `strcpy`, `strcat`, and `system`, many of which can lead to serious security vulnerabilities. These unsafe functions are present in course materials such as in lecture slides and textbooks, and even in the code provided by instructors. We also show a high correlation between the unsafe functions used by students with those used by their instructors.

CCS CONCEPTS

• General and reference → Evaluation; **• Security and privacy → Vulnerability management**; **• Social and professional topics → Computing education programs**.

KEYWORDS

Computer security education; Computer systems; Unsafe functions

ACM Reference Format:
Majed Almansoori, et al. 2020. How Secure are our Computer Systems Courses?. In *Proceedings of the 2020 International Computing Education Research Conference (ICER '20), August 10–12, 2020, Virtual Event, New Zealand.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3372782.3406266

1 INTRODUCTION

As our world's key infrastructure is increasingly reliant on information technology, the need for a skilled workforce capable of building reliable and secure software tools is more important now than ever before. Security vulnerabilities in software can lead to sensitive data theft [35], unavailability of services [24], and, even worse, disruption of critical infrastructure, such as power grids [45, 50].

Despite the dire need, in the current computer science (CS) curriculum in the United States (US), students are not introduced to computer security early on [43]. Even worse, as we show in this paper, instructors often accidentally introduce students to unsafe functions, which can easily lead to severe security vulnerabilities, via their lectures, course contents, or code skeletons provided for assignments. Effectively, students are trained to use unsafe functions without learning their security problems. As a result, our software developers are prone to introduce severe vulnerabilities in code, which attackers can exploit to compromise web services [35] and steal sensitive information [15, 40].

Studies have shown that introductory courses often overlook security vulnerabilities while teaching [47, 55]. For example, Taylor et al. [55] showed that introduction to database courses regularly teach students with vulnerable code that will lead to SQL injection attacks [9]. Taking a similar perspective, we ask, "Do other introductory courses also teach similar vulnerable code?"

In this paper, we investigate how students are taught CS introductory courses with unsafe functions and coding practices, and how they repeat similar dangerous mistakes in the code they write for programming assignments. For our analysis, we focus on a mid-level computer science course, commonly named Introduction to Computer Systems or computer systems, for short. We chose the computer systems course since it is the first course in the introductory sequence of required courses and the bedrock for teaching students subsequent computer science concepts. We also believe this course has a high potential for introducing topics related to computer security since this course focuses on low-level topics in computing, such as, call stack. The course is mostly standardized across the US and usually taught using the C or C++ programming language, including some form of an assembly language (e.g., Intel x86 Assembly). Given that, we aim to answer the following two research questions in our study:

RQ1: Do students use unsafe functions while writing code in computer systems courses?
RQ2: Are students being taught computer systems courses with unsafe functions?

To answer these questions, we gathered code samples from a number of US universities. We considered the top 20 R1 universities — universities that grant doctoral degrees and have very high research activity — based on US News rankings [42], and identify the courses equivalent to computer systems in these universities. From 16 of the 20 universities, we were able to find a matching computer systems course that is taught in C/C++ and had some publicly available course content. We downloaded the code snippets provided by instructors to their students on the course websites. We also collected code relevant for computer systems courses in these universities from public GitHub repositories. (Students often share their code on public GitHub, despite some universities discouraging students from doing so.) We collected more than 760 thousand lines of C/C++ code in this process, out of which 567.3 KLOC[1] are written by 253 students, and 193.2 KLOC are written or provided by instructors.[2]

Using a static analysis tool named Flawfinder [56], we identify the use of several dangerous functions, such as `strcpy`, `strcat`, and `atoi` in these lines of code. Despite the availability of secure alternatives, such as `strncpy` or `strncat`, such functions are still taught and used in introductory courses. Often, the main argument for teaching these functions is to reduce cognitive load on students by focusing primarily on teaching basic string operations rather than secure programming practices. Usually, the burden of teaching computer security related topics (e.g., secure programming) is saved for separate, dedicated computer security courses that students can take later in their CS curriculum.

However, our further investigation reveals that in many R1 universities in the US, students can graduate with a CS degree without taking a single computer security course since security courses are mostly offered as upper-level electives (e.g. [2, 3]). This means that students may not learn the perils of using unsafe functions or their alternative safer versions in their entire CS curriculum. As such, we are not training our software engineering workforce with adequate knowledge of safe programming. This might have severe consequences on the safety and security of the key infrastructure that relies on software.

We believe that our study highlights the problems with our computer systems courses, with respect to security, and shows the importance of the discussion of security-related topics in these courses — especially as it might be the only place where many of our CS majors learn about security. We hope that our study creates a valuable discussion among the CS education community about the importance of computer security and ways to integrate it into our current curriculum. We consider our study to be one of the first steps toward teaching secure programming practices in lower-division undergraduate CS courses.

The main contributions of our paper are the following:

(1) By analyzing more than 760 thousand lines of code written by students and instructors for computer systems courses in the US, we present the security issues that are most commonly found in these code snippets.
(2) We show that students often use unsafe functions that are often used by instructors and provided in textbooks. However, they rarely learn the security consequences behind them, and sometimes despite knowing the safer alternatives, continue using the unsafe ones.
(3) We highlight the lack of security focus in our present CS curriculum. We provide some suggestions on how to improve computer systems courses by integrating computer security.

2 RELATED WORK

Given the importance of teaching computer security to software developers, several papers have been published in the last two decades focusing on cybersecurity education and including security awareness in introductory computer science courses [32, 38].

Svabensky et al. [52] analyze 71 papers from SIGCSE and ITiCSE (no security-related papers in ICER!) and synthesize their results to summarize the current trends in cybersecurity education. They show that there are two main directions of cybersecurity research. The first direction is on teaching core security topics, such as network security, software security, and human aspects of security. The second direction is to improve students' behavior in writing secure code. Several studies, such as [21, 31, 49, 54], focused on trying to teach and encourage students to write safe code. Nevertheless, Svabensky et al. noted that the prior work rarely provides concrete, actionable suggestions for the instructors, and did not release their artifacts or datasets.

Introduce security early to students. Prior research [18, 41] has shown that introducing computer security early in the student's computer science curriculum helps students develop a security mindset and the foundation for writing secure code. Bishop and Frincke [25] describe the cruciality of improving software assurance by having students focus on secure programming practices early on and by being exposed to major but frequent security mistakes. They encourage familiarizing students with buffer-overflow vulnerability and explain the role of checklists in an aircraft scenario as a method of implementing secure code principles. The main message is that writing secure code should be one of the fundamentals taught for every computer science student.

Moving a step forward to having students write and get feedback on their code, Bishop and Orvis [18, 19] designed a *security clinic*. Students were required to submit their programming assignments to the clinic before it was due. Through the clinic, each student met with a graduate student who gave them feedback about their code's robustness. The students can change their code before they submitted their assignments for a grade. Bishop and Orvis analyzed the differences between the code submitted to the clinic vs. the final submission (for a grade). They found that, after the clinic, a majority of students understood the issues in their code better and took measures to fix them. Overall, they show that programming clinics are a viable method to reinforce writing secure code.

Irvine et al. [30, 32] took a similar approach to teach students secure coding practices and information assurance. In addition to improving the security of code, Hooshangi et al. [29] show that inculcating a security mindset also enhances students' ability to test their own programs and write robust code.

Tools for secure programming. Though security clinics (such as the one by Bishop and Orvis [18, 19]) are a great way to give students feedback, they require significant person-power to run. Also, given large — and increasing — class sizes of computer science courses, the security clinic by itself is not scalable. Zhu et al. [58] introduce the use of an interactive tool, called ASIDE, as a potential avenue for helping students learn and practice secure programming. As a plug-in for IDEs, ASIDE detects and flags potential security issues in students' code as they are writing. It then interactively assists students in fixing the issues by giving resources on those vulnerabilities. Despite some deficiencies in the tool, Zhu et al.

[1]KLOC = kilo lines of code, 1 KLOC = 1000 lines of code
[2]Researchers can contact the first author at malmansoori2@wisc.edu for access to the anonymized dataset.

conclude that ASIDE can be an effective tool for teaching security vulnerabilities.

Whitney et al. [57] extended ASIDE framework to build ESIDE that can provide nearly real-time instructional guidance and feedback with regards to secure coding practices. By studying two sets of students in an advanced web programming course, Whitney et al. show the potential of ESIDE for raising awareness in programming security. They also note that such a tool proved to be more effective when instructors incentivize its use. A similar conclusion was reached by Tabassum et al. [53] as well. They combine the two methods — the ESIDE tool and one-to-one security clinics with teaching assistants (TAs). They observe students' interaction with brief tutorials shown by ESIDE and how students discuss those issues with TAs. Tabassum et al. conclude that though ESIDE helps to raise security awareness, it alone is not sufficient; students need to be incentivized to write secure code.

Security issues in course curriculum. Though there were several approaches to include security in computer science courses, little research looked into how existing computer science courses measure up to the best security practices. Taylor and Sakharkar [55] were the first to examine whether or not SQL injection (SQLi) vulnerabilities [9] are mentioned in popular undergraduate database textbooks and course contents. Alarmingly none discusses the SQLi vulnerability in detail, despite SQLi being on the most frequent cause of data theft for decades [23]. Furthermore, two textbooks provided easily injectable example code. While Tylor and Sakharkar looked at database courses, there is a need to re-evaluate all the required introductory courses in computer science to ensure that we are not teaching insecure code to students.

Fischer et al. [26] extensively studied code snippets on Stack Overflow by scanning for code used in Android applications. They found that 15.4% of 1.3 million Android apps contained security-related code snippets from Stack Overflow. Of these, about 98% contained at least one insecure code snippet.

The missing pieces. Much of the prior security education research has been aimed towards examining or supplementing the quality of security education using resources outside the instructors' control, such as implementing a supplemental security clinic or a plugin for IDE. Although informative and useful, instructors might not have the time, expertise, and resources to implement them for their course.

Furthermore, we notice that much of this research does not directly address the root of the issue — the effects of not teaching security. Our study highlights the issues of not teaching security-related topics explicitly and how it might affect our students' code writing behavior. We found that unsafe functions are being taught or used by instructors in these courses. Students emulate their instructors and use these insecure functions when writing their own code for their class assignments, effectively perpetuating the use of these insecure functions. Moreover, we show that many of our students may graduate without even taking a single computer security course and so the poor coding practices they may have acquired while at school might hamper their professional careers.

Also, most of the prior work in security education mainly focuses on interventions to help students learn secure programming practices [25, 41, 53]. To the best of our knowledge, there is no prior work highlighting the security issues in a particular course by analyzing the thousands of lines of code snippets written by instructors and students. We believe that exposing the security problems present in instructors' and students' code will enable a much-needed conversation among CS education researchers about integrating security tightly with how we (and our students) write code, not just as an add-on to our courses.

Unsafe func.	**Reason**	**Safer alternatives / Suggestions**
`strcpy`	Can cause buffer overflow due to out of bound write. (CWE-120, CWE-121, CWE-785)	Use `strncpy`/`strlcpy`
`strcat`		Use `strncat`/`strlcat`
`(v)sprintf`		Use `(v)snprintf`
`gets`		Use `fgets`
`realpath`		Ensure output buffer is larger than `PATH_MAX`.
`memcpy`		Ensure the destination buffer is large enough.
`atoi`	Can cause integer-overflow. (CWE-190)	Use `strtol`.
`popen`, `system`, `exec*`	Can lead to OS command injection. (CWE-78, CWE-88)	Use library calls instead of calling external processes.
`getopt*`	Can cause buffer overflow and change control flow. (CWE-20, CWE-120)	Limit the length of string inputs.
`*printf`	Variable format string can allow arbitrary stack manipulation. (CWE-134)	Format string must be constant.

Figure 1: The list of flaws we considered in our study. We note why a flaw can lead to security vulnerability and how we can avoid them. We also show the associated common weakness enumeration (CWE) number in the last column.

3 BACKGROUND

We investigate an introductory level course in the computer science curriculum — computer systems — to motivate the need to teach secure programming and instill a security mindset in computer science students. Taking this course as a case study, we show that there is a dire need for teaching students to avoid certain functions that can be dangerous if misused. We give a brief background on those *unsafe functions* and why we should teach not to use them in computer systems classes. We focus on a few concrete mistakes that have caused several vulnerabilities and security breaches in the past. In Figure 1, we noted the unsafe functions we consider for this study and their secure alternatives.

Buffer overflow. The buffer overflow [4, 16] vulnerability has been known for nearly four decades [22]. The first well known exploit of stack-based buffer overflow was used in the Morris worm [44] in 1988. The buffer overflows when a (user-controlled) input larger than the size of a buffer is written to the buffer without checking the bounds. As C and C++ provide low-level access to the memory and stack, a buffer overflow can be exploited to run arbitrary code of the attacker's choice [17]. While buffer overflow can result from several insecure coding styles, some functions particularly increase the chance of inserting these buffer overflow vulnerabilities: `gets`, `strcpy`, `strcat`, and `(v)sprintf`. We should avoid these functions, and use their secure alternatives, such as `strncpy`, `strlcpy`, `strncat`, `strlcat`, `(v)snprintf`, etc. Some implementations of certain standard library C/C++ functions such as `memcpy`, `getopt`, and `realpath` had vulnerabilities in the past. These functions, therefore, should be avoided or used with caution.

Integer overflow. Similar to buffer overflow, integer overflow [6] vulnerabilities appear due to the limit on the size of integers in traditional computers. It usually occurs due to arithmetic operations that result in a number larger than the maximum size allowed, e.g., 2,147,483,647 for 32-bit signed int. A value larger than that results in "integer rounding", which is interpreted as a negative number by the computer; so 2,147,483,648 is interpreted as -2,147,483,648. A popular function that can cause integer overflow if misused is `atoi`. An attacker can exploit integer overflow vulnerabilities and control program flow by circumventing some security measurements. It is advised to avoid using `atoi`, and instead use safer alternatives such as `strtol`.

OS command injection. Functions such as `system`, `popen`, or `exec*` family of functions allow running a new process from user-provided commands or executing a command by spawning a shell. If an attacker can control the input to these functions, he can inject arbitrary commands in the inputs which will be executed by the vulnerable program [7, 8]. Therefore, `system` and `popen` should not be used at all and `exec*` should be used with caution. Many of the commands executed using these functions can be done through their equivalent library functions without running a new process. Students should be taught about those alternatives and perils of misusing these functions.

Format string vulnerabilities. Formatting functions from the `*printf` family, such as `sprintf`, `fprintf`, and `printf`, can be vulnerable if the format string is controlled by user input. An attacker can provide a crafted format string that will result in an out-of-bounds buffer read, revealing sensitive information on the buffer, or buffer manipulation, resulting in hijacking the control flow. Format string vulnerabilities [5] are normally easy to avoid by using constant formatting strings.

Security analysis. Identifying security problems can be challenging. Therefore, program analysis techniques for finding general bugs, such as static and dynamic analysis, are used to identify security issues. In static analysis, the program binary is analyzed without executing it, while in dynamic analysis, the program is executed in a strictly monitored environment to identify its behavior. Static analysis is done on the program source code, compiled byte code, or native binaries. Static analysis is often faster to execute than dynamic analysis and easier to explain to a developer.

Computer systems. Computer systems is a mid-level computer science course taught in almost every US university that grants a bachelor's degree in computer science. In this course, students learn the internal details of "how a single program runs on a computer". The key learning objectives include familiarizing students with different parts of a process's memory, such as the stack, heap, and data sections, and how they change during process execution. With a few exceptions, the course is typically taught using C/C++ and an assembly language. Being able to program in a high-level programming language is a prerequisite for this course. In some universities, students also take a course on data structures before taking computer systems.

A subset of the following topics are usually taught in most Computer Systems courses: (1) C programming, (2) assembly level programming, (3) stack, (4) dynamic memory allocators (e.g., `malloc`), (5) cache memories, (6) linking and loading, (7) virtual memory, (8) concurrency, (9) exceptional control flow, and (10) network programming. One of the widely used textbooks for this course is "Computer Systems: A Programmer's Perspective" [20]. Another more recent online textbook for the course is "Dive Into Systems" [39].

In some universities, security-related topics, such as buffer overflow and stack smashing, are taught during Computer Systems course with varying levels of depth. Nevertheless, as shown in our study, the code snippets produced by instructors and students in this course contain many security vulnerabilities.

4 METHODOLOGY

We aim to understand how often students of top US universities learn — or rather do not learn — about unsafe coding practices. To do so, we pick the computer systems course for our analysis and collect code samples written by students and instructors of the course in the top US universities. We analyze these code samples using a static analysis tool to find the use of unsafe functions. We further collect textbooks, lecture notes, and other resources used in these courses and assess their discussions about security and unsafe functions — if any is found. In this section, we describe how we collected the code samples and the static analysis tool.

Selecting universities. We consider the top 20 R1 universities (according to a US News article published in 2018 [42]) that provide bachelor's degrees in computer science for our investigation. In each of these university's curricula, we find the course that teaches computer systems. The list of universities and the course we found closest to teaching computer systems is given in Figure 2. For each of these courses, we note the course numbers, the links to the website of the most recent offering of the course, the information related to instructor contacts, course materials, such as textbooks and links to other websites, and the skeleton code used by the instructor in lectures or class assignments.

Not all courses are suitable for our study. For example, at MIT and Columbia, computer systems courses are not taught in C/C++. As the focus of this paper is to look at unsafe functions used in C/C++, we do not consider these two universities further. In Princeton University and UIUC, we could not find a course that teaches computer systems. Hereafter, we will only focus on the sixteen remaining universities for which we could gather sufficient details about the computer system courses offered.

4.1 Gathering Code

After deciding on the universities, our goal is to find code snippets written or provided by instructors as well as code written by the students during the course. While the code used by instructors will provide insights into how unsafe functions are still used during

University	Course No. and Title
Carnegie Mellon Univ. (CMU)	15-213: Intro to Computer Systems
MA Inst. of Tech (MIT)[†]	6.033: Computer Systems Engineering
Stanford Univ.	CS 107: Computer Organization & Systems
Univ. of CA, Berkeley (UCB)	CS61C: Great Ideas in Comp. Arch.
Univ. of IL (UIUC)	(No relevant course available)
Cornell Univ.	CS 3410: Comp. System Org. & Prog.
Univ. of WA (UWash)	CSE 351: The Hardware/Software Interface
Georgia Inst. of Tech (GTech)	CS 2200: An Intro to Comp. Systems & Nw
Princeton Univ.	(No relevant course available)
Univ. of Tx. (UT), Austin	CS 429: Computer Organization & Architecture
CA Inst. of Tech. (CalTech)	CS 24: Intro to Computing Systems
Univ. of Mi. (UMich)	EECS 370: Intro to Computer Organization
Columbia Univ.[†]	CSEE W3827: Fundamentals of Comp. Systems
Univ. of CA, LA (UCLA)	CS 33: Intro to Computer Organization
Univ. of WI (UW–Madison)	CS/ECE 354: Intro to Comp. Systems
Harvard Univ.	CS 61: Sys. Prog. & Machine Org.
Univ. of CA, SD (UCSD)	CSE 30: Comp. Org. & Systems Prog.
Univ. of MD, CP (UMD)	CMSC 216: Intro to Computer Systems
Univ. of PA (UPenn)	CIS 240: Intro to Computer Systems
Purdue Univ.	CS 252: Systems Programming

[†] Does not teach the course using C/C++.

Figure 2: Top 20 universities in the US that offer bachelor's in computer science (or equivalent) according to US News [42]. We also note the courses we found that are closest to a computer systems course in their undergraduate curricula. We do not consider 4 of the top-20 universities.

teaching introductory systems courses, students' code will show how students reuse similar vulnerabilities in their code.

Code from course websites. Instructors often use code snippets in their study materials or provide code skeletons for home or in-class assignments. We collected such code from the course websites. For many universities, such code examples are publicly available; for others, they are kept behind university login. For five universities, we could not find any code as they require special permission to access. We contacted instructors who teach these courses, asking them for a copy of these code snippets. Unfortunately, while we received a response from 4 instructors, none of them were willing to share their solution code with us over the concern of the code being leaked to the public.[3] Even after assuring that we will not publicly release any of the code, the instructors were still reluctant to share their code snippets or skeletons, stating that those were written without security in mind.

Code from git repositories. Students often upload their code, including assignment solutions, to code repositories hosting services, such as GitHub [10], Bitbucket [1], and GitLab [11], to publicize their work. To understand the use of unsafe functions by students, we turn to find code written by students for their computer systems course available on public code hosting services.

We tried using Google's site-specific search option [13] to find student's code in all three code hosting services mentioned above. We used the name of the university and the course number as search queries. Although such search returned many results, only a few of them were relevant to the courses we are interested in.

Therefore, we started searching using the search tools provided by those three code hosting services. We kept our queries simple and short to minimize noisy results. For each university, we started the search using the course number only. Whenever we received too many irrelevant results — such as when two universities share the same course number — we added the name of the university or the name of the course to the query to improve the search results. This method resulted in several relevant code repositories in GitHub, but none on Bitbucket or GitLab.

To expand our dataset of code, we also utilized the 'search-by-code' functionality provided by GitHub [14]. From the repositories we already collected, we pick unique lines of code (e.g., function declarations) to find more similar repositories. We removed repositories that do not contain C/C++ code using GitHub filters.

Code cleaning. Many of the repositories returned by these search techniques were not relevant to the computer systems course. So we further filtered the code repositories to remove false positives. For filtering, we manually checked whether any file (esp. the README file) in the repository mentions the university's name or the corresponding course name/number. If so, we include the repository to our dataset. Sometimes, the repository did not include any such information; in those cases, we check whether the projects are similar to the repositories we already included in our dataset. We also checked the user's GitHub profile to determine if the user attended the university we are searching for. If we were not able to conclude that a repository is for a computer systems course and belongs to a user from one of the universities we are interested in, then it is excluded from our dataset. Further, we removed the repositories that only contain skeleton code provided by instructors.

We found 295 repositories from 253 students. We assumed each user account in GitHub corresponds to a distinct student. Though, in theory, students can create multiple GitHub user accounts and upload their code to multiple user accounts, we believe this will be unlikely to happen in practice. We downloaded all such repositories. Some users put the code for all of their assignments in one repository, while others created separate repositories for each assignment. Some repositories even contain code from other courses; we manually went over the repositories and removed them.

As we are interested in finding the use of unsafe functions in C/C++ code, we only considered files with `.c`, `.cpp`, and `.cc` extensions, ignoring all other files, including header (`.h`) files. We assumed that header files primarily contain function declarations, and do not invoke any function (including unsafe functions). We noticed that some repositories contain folders named `include` and `cgi-src` that only contain standard library functions. This code is unlikely to be written by the instructors or students; so, we removed those folders and their contents from our dataset. After filtering, we have 740,114 lines of code in 7,208 files.

Code attribution. Note, students often start from skeleton code provided by their instructors. As such, students' and instructors' code can be present in the same file. Therefore, we separate chunks of code in a file and attribute them to written by students or provided by instructors.

[3]The assignment code is often reused over multiple years. The instructors would have to create new assignments if the code is posted publicly.

All the code we downloaded from the course websites are attributed to the instructors. The code downloaded from GitHub is attributed in the following way. We assume students' code is likely to be distinct from each other. Therefore, all the files present in multiple user's repositories are considered as *not* written by students, and are attributed to instructors. For each of the remaining files in our dataset, we considered all code chunks of l lines and checked if they are present in multiple students' repositories, if so, those chunks of code are attributed to instructors. We create validation data by manually going over all the lines of code of a randomly chosen university and flagging them as written by instructors or by students. Then we tested the accuracy of our algorithm based on different values of l. We found $l = 10$ provided the best result. Any code that is not attributed to instructors is assumed to be written by students.

Eventually, we have 193,239 lines of code from instructors and 567,342 lines of code from students. In the next section, we show how we analyzed the security of these code files.

4.2 Static analysis.

We analyzed the security of all code we collected from course websites and GitHub using a simple static analysis tool called Flawfinder [56]. Flawfinder is a simple yet powerful static analysis tool that searches for a predefined list of unsafe functions in C/C++ code, without executing it. Admittedly the precision and recall in finding security vulnerabilities of Flawfinder is worse than other state-of-the-art static analysis tools [36]. However, there are two main benefits of Flawfinder. First, unlike many other tools, Flawfinder is fast and runs on code snippets as well as on code with compilation errors. This is crucial as much of the code snippets we collected do not compile. Second, by focusing on a handful of egregiously unsafe functions, such as `strcpy`, we hope to provide more actionable advice to the instructors. We examined other static analysis tools as well, such as Infer [12] and Graudit [37], but none could match Flawfinder's performance and ease of use.

We noticed that Flawfinder does not correctly distinguish function declaration or definition from function invocation. We, therefore, applied a heuristic on top of Flawfinder results to ignore function definitions and descriptions. Besides, Flawfinder, in its default configuration, provides a large number of warnings, many of which might not lead to security flaws. Instead of giving too many false alarms, we are focusing on a small number of unsafe functions that should be avoided.

Flaws re-evaluation. Flawfinder uses levels to determine the severity of a flaw: level 1 to level 5, where level 5 is the most severe, and level 1 is the least severe. These flaws are typically based on use of unsafe functions. We re-evaluated and simplified the levels to make it more suitable for analyzing computer systems courses. We regrouped the different flaws identified by Flawfinder into three levels: `L0`, `L1`, and `L2`.

Level 2 (`L2`) flaws can cause buffer overflow [4] or code injection [7, 8] vulnerability. This category of flaws uses functions such as `strcpy`, `strcat`, `(v)sprintf`, `system`, and `gets`. These functions therefore should be avoided.

We used level 1 (`L1`) to denote the flaws that are less security critical — less likely to be a cause of a security vulnerability — than `L2`. These functions should be avoided if possible, and one should be careful while using them. This category includes `atoi`, `exec*`, `memcpy`, etc. Flawfinder also warns about certain use of functions such as `scanf`, `access`, `readlink`, and `printf`. We classified these flaws as level zero (`L0`) and decided to ignore them for the purpose of this study. We focus on `L1` and `L2` flaws since they are known to cause severe security vulnerabilities in practice. In Figure 3, we record the list of `L1` and `L2` functions. The security problems caused by them and their safer alternatives are given in Figure 1.

Level	Unsafe func.	Student	Instructor	Total	Flaw category
L1	`atoi`	738	1,099	1,837	integer overflow
	`memcpy`	335	392	727	buffer overflow
	`getopt*`	69	283	352	buffer overflow
	`exec*`	74	82	156	code injection
	`(v)snprintf`	18	17	35	format string
	`realpath`	10	0	10	buffer overflow
	`popen`	2	7	9	code injection
L2	`strcpy`	693	1011	1,704	buffer overflow
	`(v)sprintf`	442	908	1,350	buffer overflow
	`strcat`	692	361	1,053	buffer overflow
	`system`	19	76	95	code injection
	`gets`	7	2	9	buffer overflow
	Total	3,099	4,238	7,337	

`exec*` includes `execvp`, `execlp`, `execl`, `execle`, and `execv`.
`getopt*` includes `getopt` and `getopt_long`

Figure 3: Count of `L1` and `L2` flaws across all universities and their categories (the last column) as noted by Flawfinder. We combine some family of functions into one, such as `exec*` and `getopt*` for ease of presentation.

5 RESULTS

We analyze the code we collected using Flawfinder [56]. In this section, we report what types of errors we find in code written by students and the code provided by instructors. We show that instructors teach computer systems course with several unsafe functions and students frequently use those functions. Moreover, we found that, in some universities, students are aware of safer alternatives to these functions, but still use the safe functions in conjunction with unsafe functions or misuse them, rendering them equally insecure.

5.1 Use of Unsafe Functions

We collected 760,581 lines of code produced by students and instructors of 16 of the top 20 R1 universities in the US. We ran Flawfinder with modified flaw levels (as described in Section 4.2) on the code.

We present the frequency of different flaws in Figure 3 as observed in students' and instructors' code. We found the use of level two (`L2`) unsafe functions — which can be very dangerous to be used in production code — are used more frequently than level one (`L1`) functions. About 57% of all the unsafe functions we found belong to the `L2` category. The most widely used `L2` flaw among students and instructors is `strcpy`, used 1,704 times in our whole code dataset, though instructors used this function more frequently than students. Students of all universities except U7, U9, U12, and U13 used `strcpy` at least once in their code, and instructors of all universities but U3 and U16 used `strcpy`. Other `L2` errors, such

Univ.	Students' code				Instructors' code			
	L1	L2	KLOC	fKLOC	L1	L2	KLOC	fKLOC
U1	134	**337**	70.7	6.7	324	**706**	**30.9**	33.3
U2	46	69	41.4	2.8	112	57	**20.4**	8.3
U3 *	19	**380**	16.9	**23.6**	0	34	0.3	113.3
U4	70	**392**	85.9	5.4	142	151	**11.5**	25.5
U5	195	77	58.7	4.6	228	65	**28.9**	10.1
U6 *	43	163	31.5	6.5	1	34	1.6	21.9
U7	7	0	31.8	0.2	153	**263**	**18.6**	22.4
U8 *	57	**233**	50.1	5.9	124	**561**	**13.6**	50.4
U9	102	0	18.2	5.6	91	77	5.3	31.7
U10	221	73	11.8	**24.9**	91	74	0.7	235.7
U11	40	19	60.6	1.0	152	151	**29.3**	10.3
U12	73	1	26.4	2.8	366	70	**15.1**	28.9
U13	98	0	21.8	4.5	36	14	2.3	21.7
U14	101	58	18.8	8.5	29	80	4.7	23.2
U15 *	19	18	12.1	3.1	27	21	7.9	6.1
U16 *	21	33	10.7	5.0	4	0	2.1	1.9
Total	1,246	1,853	567.3	5.5	1,880	2,358	193.2	21.9

* No instructor code snippet is available publicly.

Figure 4: The frequency of level one (L1) and two (L2) unsafe functions in the code written by students and provided by instructors. We also note the kilo lines of code for each university we found for students and instructors. The columns fKLOC denote the flaws per KLOC.

as strcat, system, and (v)sprintf, are also used quite frequently by students and instructors alike.

Among the L1 flaws, atoi was the most frequently used. The function atoi is used to convert a string into an integer. An attacker can provide a crafted string that will overflow the integer value [6] and force the code to deviate from the legitimate control flow. Other prevalent L1 flaws include memcpy and getopt, both of which can lead to a buffer overflow if used in unsafe ways.

In our code dataset, we found 3,126 instances of L1 flaws and 4,211 instances of L2 flaws. We show the number of flaws we found for each university in Figure 4, broken down by the code written by students and the code provided by instructors.

Flaws in instructors' code. Instructors frequently use unsafe functions in their lectures or assignment code skeletons. Though we have fewer code written by instructors, we found a much higher proportion of flaws in instructor code. We collected about 193.2 KLOC that is provided by instructors. The modified Flawfinder flagged about 22 flaws (L1 or L2) on an average per KLOC of instructors' code. In U1, U7, and U8, we have seen more than 200 invocations of L2 functions for each university. We found that the instructors in all the universities, except U16, have used at least two different unsafe functions.

Not every invocation of an unsafe function leads to a vulnerability. To understand whether these flaws are exploitable, we randomly sampled 20 strcpy flaws in instructor code flagged by Flawfinder and analyzed if an attacker can exploit them. We found that at least 8 out of the 20 flaws (40%) we evaluated can be exploited for stack smashing attacks [17]. Only 8 of them (40%) were used in a way that can be safe. For example, by ensuring that the source string is constant: strcpy(dst, "Hello").

Flaws in students' code. We had more than 560 thousand lines of code written by 253 students; median 2 KLOC per student. Some

```
int
main(int argc, char** argv) {
   char *pointer;
   pointer = malloc(SIZE);
   strpcy(pointer, argv[1]);
   ...
}
```

(a) Student code

```
int
main(int argc, char** argv) {
   ...
   char buffer[20];
   strcpy(buffer, argv[1]);
   ...
}
```

(b) Instructor code

Figure 5: Examples of strcpy in instructor and student code

Univ.	# Students	# L2		KLOC
		Avg	Median	Median
U1	21	16	**5**	4.7
U2	22	3	0	1.8
U3	12	32	**3**	1.2
U4	21	19	**12**	4.5
U5	22	3	1	3.8
U6	10	16	**13**	3.0
U8	14	17	0	2.8
U10	11	7	**7**	1.2
U14	10	6	**11**	1.9

Figure 6: Table shows the number of students (2nd column) in our dataset, the average and median number of times L2 unsafe function are used (3rd and 4th column), and the median KLOC (5th column) for universities where students used L2 unsafe functions at least 50 times.

students had significantly more lines of code than others. Some students had multiple projects on the GitHub code repositories, while others had only one or two. Despite having a lot more lines of code than instructors, we found significantly fewer instances of L1 and L2 unsafe function usage in student's code than instructors.

The distribution of flaws in student's code varies widely. For some universities, such as U7, U9, and U13, we did not find any use of L2 unsafe functions by students. While in U1, U3, U4, U6, U8, and U10, students used L2 unsafe functions more than four times every one thousand lines of code. The cumulative usage of strcat and strcpy in students' code is very similar to that used in instructors' code. Instructors use sprintf more than students, while students use strcat quite frequently. All of these functions can cause a buffer overflow.

We analyzed students' use of strcpy to compare against that of instructors. We pick 20 random instances of strcpy in student code, and manually analyzed for security issues. We found that 8 (40%) of the 20 invocations are vulnerable. Only 6 (30%) uses are appropriate in terms of security. Two representative vulnerable examples from students' and instructors' usage of strcpy are given in Figure 5. The student code (on the left) is vulnerable to overflow on heap, and the instructor code (on the right) is vulnerable to overflow on the stack, which is even worse.

The distribution of usage of unsafe functions is not uniform. At U7, the rate of use of unsafe functions is less than one per KLOC, whereas, for U3 and U10, it is almost 25 per KLOC. If we focus only on L2 flaws, four universities (highlighted in Figure 4) have more than 200 instances at a rate of over four flaws per KLOC.

In Figure 6, we show the median number of L2 errors per university for universities with more than 50 instances of L2 unsafe function invocation. As can be seen in the figure, 50% of students in U4, U6, and U14 made more than 11 level two (`L2`) flaws. In U2, U5, and U8, many students had less than one thousand lines of code, while others had a significantly large amount of code, which is why the median is below 1, but have high number of `L2` flaws.

Next, we see how similar the usage of unsafe functions is between students and instructors.

5.2 Correlation of Flaws

We hypothesize that students often use these unsafe functions because instructors use them in classes or assignment code snippets. To see whether this hypothesis is true, that is, do students make similar mistakes as their instructors, we compute the similarity between the flaws we see in instructors' code with the ones in students' code. We do not look into the fine-grained usage patterns of these functions, for example, whether students replicate a particular type of use of an unsafe function or not. Our approach also looks at correlation and does not provide causation — we cannot say beyond a reasonable doubt that instructors are the sole reason students learn and use unsafe functions.

We compute the similarity between student and instructor code in the following way. For each university, we count the number of times students and instructors use each unsafe function (mentioned in Figure 3) and note them as a vector. So, for each university, let $\vec{S_j}$ and $\vec{I_j}$ be the number of times j-th unsafe function is used by students and instructors, respectively. Then we compute the similarity between the usage of unsafe functions between instructors and students as the dot product of the vectors $\vec{S}$ and $\vec{I}$, $\mathrm{Sim}(\vec{S}, \vec{I}) = \vec{S} \cdot \vec{I} = \frac{\sum_j \vec{S}_j \cdot \vec{I}_j}{\|\vec{S}\| \cdot \|\vec{I}\|}$, where $\|\vec{a}\|$ denote the quadratic norm of the vector $\vec{a}$. The similarity score is always between $[0, 1]$. A similarity score of 0 means they are not similar at all, while a score of 1 means they are completely similar. We compute this similarity for each university. The similarities between students' and instructors' code for each university is shown in the rightmost column of Figure 7. We also show the most frequent three flaws by students and instructors for each university in the same figure.

We found that across all universities, the types and rates of flaws done by students and instructors are quite similar. Interestingly for U1, U5, U11, and U12, the similarity score is above 0.9, and students and instructors in those universities have used unsafe functions more than 50 times. For U3, U6, U10, U13, and U16 we do not have enough code for instructors to make any conclusive remark.

We cannot determine the causality of student's use of unsafe functions based on this similarity analysis. For example, it might be that in the general population of developers, the rate of use of unsafe functions is similar to what we are seeing in our restricted sample of developers — students and instructors in computer systems course in the US. However, as we do not see a uniformly high rate of similarity across all universities, and as the rate of use of unsafe functions by students differ at different universities, it is probably the case that instructors are influencing the choice of unsafe functions that students use. Nevertheless, we need more studies to conclude this hypothesis: the use of unsafe functions by instructors is the reason why students use those functions.

University	Student flaws		Instructor flaws		Similarity
U1	`sprintf:`	162	`sprintf:`	355	**0.98**
	`strcpy:`	106	`strcpy:`	263	
	`atoi:`	81	`atoi:`	204	
U2	`sprintf:`	35	`atoi:`	82	0.72
	`memcpy:`	26	`sprintf:`	26	
	`strcpy:`	22	`strcpy:`	22	
U3	`strcat:`	244	`strcat:`	32	**0.94**
	`sprintf:`	78	`sprintf:`	2	
	`strcpy:`	58			
U4	`strcat:`	202	`exec*:`	68	0.63
	`strcpy:`	112	`sprintf:`	65	
	`sprintf:`	77	`strcpy:`	61	
U5	`memcpy:`	160	`memcpy:`	161	**0.95**
	`strcpy:`	65	`getopt:`	39	
	`exec*:`	18	`strcpy:`	34	
U6	`strcpy:`	127	`strcpy:`	14	0.76
	`strcat:`	22	`system:`	12	
	`exec*:`	22	`strcat:`	8	
U7	`atoi:`	6	`sprintf:`	120	0.52
	`getopt:`	1	`atoi:`	88	
			`strcpy:`	76	
U8	`strcat:`	137	`sprintf:`	222	0.83
	`sprintf:`	52	`strcpy:`	176	
	`strcpy:`	44	`strcat:`	156	
U9	`atoi:`	100	`strcpy:`	77	0.67
	`getopt:`	2	`atoi:`	71	
			`getopt*:`	20	
U10	`atoi:`	220	`atoi:`	91	**0.94**
	`strcpy:`	73	`strcpy:`	74	
	`memcpy:`	1			
U11	`atoi:`	23	`strcpy:`	116	**0.93**
	`strcpy:`	15	`atoi:`	97	
	`memcpy:`	9	`getopt*:`	29	
U12	`atoi:`	56	`atoi:`	303	**0.99**
	`memcpy:`	11	`strcpy:`	39	
	`getopt*:`	6	`memcpy:`	37	
U13	`atoi:`	90	`atoi:`	22	0.83
	`memcpy:`	8	`strcpy:`	12	
			`memcpy:`	10	
U14	`memcpy:`	58	`strcpy:`	44	0.73
	`strcpy:`	34	`sprintf:`	24	
	`atoi:`	32	`memcpy:`	16	
U15	`memcpy:`	9	`atoi:`	14	0.77
	`sprintf:`	8	`sprintf:`	8	
	`strcpy:`	6	`system:`	8	
U16	`strcpy:`	31	`atoi:`	4	0.00
	`sprintf:`	10			
	`getopt*:`	5			

Figure 7: Table shows the top 3 flaws in students' and instructors' code and the similarity between the use of unsafe functions by students and instructors (right column).

5.3 Unsafe use of functions

Unsafe functions clearly should be discouraged, and safer alternatives should be used to improve the safety of user code. However, using safe functions is not sufficient to write secure code. We found several instances where safer alternative functions are used in unsafe ways, effectively voiding the security benefits. For example, `strncpy` is often advised to be used instead of `strcpy`. However,

Textbook	# Uni.	Discuss sec.	Safe code
Comp. Systems: A Programmer's Persp. [20]	8	**Yes**	**No**
The C Programming Language [33]	6	No	No
Comp. Org. & Design: The HW/SW Interface [46]	3	No	No
C: A Reference Manual [27]	2	Yes	No
Dive Into Systems [39]	2	**Yes**	**Yes**
Digital Design and Computer Arch. [28]	1	Yes	No
C Programming: A Modern Approach [34]	1	No	Yes
Advanced Programming in the UNIX Env. [51]	1	Yes	No

Figure 8: Textbooks used in computer systems courses. The third and the fourth columns show whether or not buffer overflow is discussed, and any unsafe functions are used in code snippets in the textbook. We only consider textbooks used by at least two universities and textbooks that discuss security or do not use unsafe functions.

we found some use cases where students misuse it. We give an example from our dataset below:

```
int main(int argc, char* argv[]) {
   char buf[SIZE];
   strncpy(buf, argv[1], strlen(argv[1]) + 1);
   ...
```

In this code example, an attacker can still cause a buffer overflow by supplying an `argv[1]` that is larger than `SIZE`.

In several instances we found students use both `strcpy` and `strncpy` in the same file. This means some students are aware of the safer alternative but unaware of their security benefits. Therefore, it is not enough to tell students to use the safer alternative, but it is also necessary to teach (a) why the unsafe functions should be avoided and (b) how to write safer code. This can be done in the computer systems course as students learn about the logical execution of a single program in computer systems. They also learn about the memory layout of a running program, including stack and heap. Therefore, students can be introduced to the basics of stack smashing and how it can be avoided. This class will be a great place to teach students about the problem of writing data into a buffer without properly checking the bounds.

5.4 Class Resources

In addition to the instructors' code, we also examined textbooks and lecture slides (when available) used for these courses. We examined twelve textbooks recommended and required by these universities, checking for the use of unsafe functions, warnings against buffer overflow, and a dedicated security-related section. As seen in Figure 8, only five textbooks warned against buffer overflow as a security issue, and only two textbooks had a dedicated section on security (highlighted in Figure 8). We found that even the textbooks that warn against certain unsafe functions such as `strcpy` and `sprintf`, do not always offer alternatives such as `strncpy` or `snprintf`. Furthermore, some textbooks continue using unsafe functions throughout many of their code snippets, even after mentioning that these functions are unsafe to use.

Our assessment showed that Dive Into Systems [39] is the only textbook that discusses buffer overflow in detail and does not have any insecure code snippet. One textbook [34] did not address security but introduced the unsafe functions as functions that could lead to "undefined behavior" and explained how to use safer alternatives such as `strncpy`. According to the textbook, undefined behavior could lead to a segmentation fault or some garbage value, but it does not list their security issues.

Similarly, we examined the lecture slides and notes provided by the universities. Unfortunately, slides and other resources for U3, U6, U11, and U15 were inaccessible. Therefore, we only examined the notes from the remaining 12 universities by looking for examples containing unsafe functions and discussions about security-related topics. We found that only five universities discussed buffer overflow, unsafe functions, or other security issues (U2, U5, U7, U14, and U16). Although some of the universities introduced safer alternatives, they kept using the unsafe versions in the slides throughout the course.

We found U1, U7, U13, and U15 had a lab about exploiting a buffer overflow vulnerability to control the program's flow. The benefit of such a lab can be seen in the low usage of `L2` functions by students of U7, U13, and U15 (see Figure 4). For U1, we see a high rate of `L2` function usage, however we could not determine why it is so; further investigation is needed. Nevertheless, we envision such hands-on experience should be used to teach students about security issues with unsafe functions.

We noticed that only for U7 and U9, students did better than instructors, so we investigated that further by analyzing the nature of the programming assignments. For U9, students' projects focused on using pointers and structures, understanding the cache and CPU, memory allocation, and getting familiar with x86 Assembly language. These assignments did not require the use of `L2`. Thus, we could not conclude that students are actually aware of these unsafe functions because the assignments do not require using them. Unfortunately, we were not able to examine the assignments of U7 since the details of projects are not publicly available.

6 DISCUSSION

Through our analysis in Section 5, we show that (1) students in introductory computer systems courses often use unsafe functions that can lead to severe security vulnerabilities; and (2) similar usage of unsafe functions is present in the code provided by instructors as well as in many widely used textbooks. In this section, we first give some suggestions on how to improve the status quo, and with a discussion about the limitations of the study.

How to improve? Currently, students are not being primed with how to write secure and robust code, and, even worse, they learn from vulnerable code examples. To tackle this problem, we need a multi-pronged approach.

Teach why avoid unsafe functions. First, we must ensure instructors and textbooks do not use unsafe functions without warning about them or explaining their security implications. While avoiding the unsafe functions is desirable, we argue that it is not sufficient just to teach the safe alternatives of unsafe functions; students must be taught *why* certain usages of certain functions are unsafe. Without such knowledge, students might use safe alternatives in an insecure way, as shown in Section 5.3. Finally, incorporating fruitful discussion about security in the existing course materials can be challenging, especially when ensuring the course load remains reasonable for students and instructors. Moreover, discussing unsafe functions and their risk of leading to vulnerabilities can be an important way to instill a security mindset early in students' computer science education.

Update textbooks. Textbooks affect the functions students use. We found very few instances of `gets` functions in the whole code dataset. Possibly, this is because textbooks do not use them at all and discuss buffer overflow vulnerabilities due to `gets`. Such a demonstration of security vulnerability helps instructors and students avoid the function. Other unsafe functions are not discussed with equivalent details, and therefore, despite being equally dangerous, they are still used widely in textbooks, instructors' code snippets, and by students.

Training Instructors. Instructors use unsafe functions quite frequently. Though some of those invocations might not be vulnerable, it is unclear if the subtleties in usage are discussed in the class (and students follow them). Moreover, instructors sometimes do not teach safe alternatives to reduce the cognitive overload — because safe alternatives of unsafe functions tend to be more complicated to use. Therefore, it is more important to discuss security in the computer systems course to help students understand the security issues with unsafe functions and learn to avoid them. Right now, security is discussed cursorily or not discussed at all in computer systems courses in many universities. One possible reason instructors do not talk about security in introductory classes might be due to the lack of expertise in computer security. Therefore, despite having decades long research on security education in introductory computer science courses, only few are actually implemented in schools. We need to create training materials and workshops for instructors so that they can incorporate security topics in their courses.

Incentivize secure coding. The functionality of a program is typically the primary concern of students while writing code for their assignments. Given that students are points-driven, we must dedicate certain points to incentivize students to make sufficient effort to avoid obvious security issues in their code. This could be done by penalizing a certain percentage of the total score for potential security flaws found in the code. Anecdotal evidence shows that students' code quality improves after being penalized for submitting code that fails linting tests. For security checks, tools such as ASIDE [58] or Flawfinder can be repurposed to help create similar tests to encourage students write secure code.

We also highlight the fact that in all of the top 20 R1 universities, security courses are saved for higher-level electives. While we could not gather data about what fraction of students graduate without taking a security course, it is possible that many students might not learn the perils of using unsafe functions or how to write secure code by the time they graduate with a computer science degree. Therefore, introductory courses must inform students about the basics of writing secure code.

Finally, tools such as Flawfinder [56] or LibFuzzer [48] could be used to avoid the mistakes we discussed in this paper. However, such tools are neither precise nor exhaustive. Therefore, developers should be trained with a security mindset and safe programming practices. Some might argue that security experts can deal with vulnerable programs written by developers; however, it is safer to avoid having exploitable applications in the first place. If developers are more aware of security, then security experts will be able to focus on more complicated vulnerabilities rather than simple ones that are usually missed.

Limitations. Our study makes multiple assumptions, which are possibly the key limitations of the study. We collected students' code samples that are publicly posted on GitHub. This might not be a good representation of the class of students taking computer systems course in a school. Moreover, the course names on GitHub could be erroneous. For example, students can post arbitrary code in the name of the computer systems course. Though our manual filtering did not find any such occurrences, a more authentic source of code might be better. We have very little code provided by instructors for many universities. More collaboration with instructors would help identify the key issues and address them effectively.

We show a high correlation between the usage of unsafe functions used by instructors and by students. However, our analysis does not show any causal relationship — why do students use unsafe functions. This could be due to considering security secondary to functionality while writing code. Such a "bolt-on security" attitude is known to provide poor security, and we need to find ways to ensure students are trained to take a "built-in security" approach. We highlight some of the key problems, but it is still unclear how to incorporate them into the course content (besides just to stop using unsafe functions). A more longitudinal study with interventions can help us find strategies to teach students about unsafe functions.

Also, we looked at a small number of egregious unsafe functions. Not using those unsafe functions is a step in the right direction, but it does not mean the code will be robust and free from all security issues. We only focused on computer systems course; however, similar security flaws might be present across different courses in the CS curriculum. As computer security spans across almost all areas of computer science, future work should investigate how other courses include discussion of computer security in their course materials as well as adhere to best security practices.

7 CONCLUSION

We analyzed more than 760 thousand lines of code written by students and instructors for computer systems courses in the US. We found thousands of invocations of unsafe functions such as `strcpy`, `strcat`, and `system` in the code dataset. Evidently, students and instructors alike do not consider security implications while writing code. Moreover, we found students use similar types of unsafe functions that their instructors use. We highlight two core issues with our CS curriculum: (1) students are not taught the security implications of using unsafe functions, and (2) by frequently using unsafe functions, instructors and textbooks are passively teaching students to use them. Instructors favor focusing on the functionality in computer systems courses, (possibly) saving the computer security related topics for a separate course. However, in all top 20 universities we studied, students can graduate with a CS major without taking any computer security course. So, at this point, many in our current software developer workforce are being trained to use unsafe functions without learning their security implications and can graduate without taking a formal security course. This is a threat to our key infrastructures that rely on developers writing safe and secure code.

REFERENCES

[1] Bitbucket. https://bitbucket.org/product/.

[2] CS courses at UC San Diego. https://ucsd.edu/catalog/courses/CSE.html.

[3] CS courses at UW-Madison. https://guide.wisc.edu/courses/comp_sci/.

[4] CWE-121: Stack-based buffer overflow. https://cwe.mitre.org/data/definitions/121.html.

[5] CWE-134: Use of externally-controlled format string. https://cwe.mitre.org/data/definitions/134.html/.

[6] CWE-190: Integer overflow or wraparound. https://cwe.mitre.org/data/definitions/190.html/.

[7] CWE-78: Improper neutralization of special elements used in an os command ('os command injection'). https://cwe.mitre.org/data/definitions/78.html.

[8] CWE-88: Improper neutralization of argument delimiters in a command ('argument injection'). https://cwe.mitre.org/data/definitions/88.html.

[9] CWE-89: Improper neutralization of special elements used in an SQL command ('sql injection'). https://cwe.mitre.org/data/definitions/89.html.

[10] Github. https://github.com/about.

[11] Gitlab. https://about.gitlab.com/.

[12] Infer. https://fbinfer.com.

[13] Refine web searches. https://support.google.com/websearch/answer/2466433?hl=en.

[14] Searching code. https://help.github.com/en/github/searching-for-information-on-github/searching-code.

[15] 2017 Cybersecurity Incident & important consumer informationlist of data breaches. https://www.equifaxsecurity2017.com/consumer-notice/, 2018.

[16] CWE-122: Heap-based buffer overflow. https://cwe.mitre.org/data/definitions/122.html, 2020.

[17] One Aleph. Smashing the stack for fun and profit. *http://www. shmoo. com/phrack/Phrack49/p49-14*, 1996.

[18] Matt Bishop. A clinic for "secure" programming. *IEEE Security and Privacy*, 8(2):54–56, 2010.

[19] 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.

[20] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 2. Prentice Hall Upper Saddle River, 2003.

[21] Sam Chung, Leo Hansel, Yan Bai, Elizabeth Moore, Carol Taylor, Martha Crosby, Rachelle Heller, Viatcheslav Popovsky, and Barbara Endicott-Popovsky. What approaches work best for teaching secure coding practices. In *Proceedings of the 2014 HUIC Education and STEM Conference*, 2014.

[22] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[23] Dancho Danchev. Reports: SQL injection attacks and malware led to most data breaches. https://www.zdnet.com/article/reports-sql-injection-attacks-and-malware-led-to-most-data-breaches/, 2010.

[24] Fred Donovan. Healthcare industry takes brunt of ransomware attacks. https://healthitsecurity.com/news/healthcare-industry-takes-brunt-of-ransomware-attacks/.

[25] Bishop Fischer and Deborah Frincke. Teaching secure programming. In *IEEE Security and Privacy*, pages 54–56, 2005.

[26] Felix Fischer, Konstantin Bottinger, Huang Xiao, Christian Stransky, 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.

[27] Samuel P Harbison. *C: a reference manual.* Prentice Hall, 2002.

[28] David Harris and Sarah Harris. *Digital design and computer architecture.* Morgan Kaufmann, 2010.

[29] 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.

[30] Cynthia E Irvine. What might we mean by "secure code" and how might we teach what we mean? In *19th Conference on Software Engineering Education and Training Workshops (CSEETW'06)*, pages 22–22. IEEE, 2006.

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

[32] Cynthia E Irvine, Michael F Thompson, and Ken Allen. Cyberciege: an extensible tool for information assurance education. Technical report, Naval Postgraduate School Monterey Ca Center For Information Systems, 2005.

[33] Brian W Kernighan and Dennis M Ritchie. *The C programming language.* 2006.

[34] K. N. King. *C Programming: A Modern Approach, Second Edition.* W.W. Norton & Company, 2008.

[35] Timothy B. Lee. Dnc email leak explained. https://www.vox.com/2016/7/23/12261020/dnc-email-leaks-explained, 2016.

[36] Rahma Mahmood and Qusay H Mahmoud. Evaluation of static analysis tools for finding vulnerabilities in java and c/c++ source code. *arXiv preprint arXiv:1805.09040*, 2018.

[37] E Marcussen. Graudit. https://github.com/wireghoul/graudit/.

[38] Stefanie A Markham. Expanding security awareness in introductory computer science courses. In *2009 Information Security Curriculum Development Conference*, pages 27–31, 2009.

[39] Suzanne J. Mathews, Tia Newhall, and Kevin C. Webb. Dive into systems. https://diveintosystems.cs.swarthmore.edu/.

[40] Elinor Mills. List of data breaches. https://en.wikipedia.org/wiki/List_of_data_breaches, 2020.

[41] Kara Nance. Teach them when they aren't looking: Introducing security in cs1. *IEEE Security and Privacy*, 7(5):53–55, 2009.

[42] U.S. News. Best computer science schools. https://www.usnews.com/best-graduate-schools/top-science-schools/computer-science-rankings, 2018.

[43] Linda Null. Integrating security across the computer science curriculum. *Journal of Computing Sciences in Colleges*, 19(5):170–178, 2004.

[44] Hilarie Orman. The morris worm: A fifteen-year perspective. *IEEE Security & Privacy*, 1(5):35–43, 2003.

[45] Amer Owaida. European power grid organization hit by cyberattack. https://www.welivesecurity.com/2020/03/12/european-power-grid-organization-entsoe-cyberattack/, 2020.

[46] David A Patterson and John L Hennessy. *Computer Organization and Design ARM Edition: The Hardware Software Interface.* Morgan kaufmann, 2016.

[47] Huwida E Said, Mario A Guimaraes, Zakaria Maamar, and Leon Jololian. Database and database application security. *ACM SIGCSE Bulletin*, 41(3):90–93, 2009.

[48] Kostya Serebryany. Libfuzzer: A library for coverage-guided fuzz testing (within llvm).

[49] Ambareen Siraj, Blair Taylor, Siddarth Kaza, and Sheikh Ghafoor. Integrating security in the computer science curriculum. *ACM Inroads*, 6(2):77–81, 2015.

[50] Blake Sobczak. Report reveals play-by-play of first u.s. grid cyberattack. https://www.eenews.net/stories/1061111289/, 2019.

[51] W Richard Stevens and Stephen A Rago. *Advanced programming in the UNIX environment.* Addison-Wesley, 2008.

[52] Valdemar Švábenskỳ, Jan Vykopal, and Pavel Čeleda. What are cybersecurity education papers about? A systematic literature review of sigcse and iticse conferences. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 2–8, 2020.

[53] 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.

[54] Blair Taylor and Shiva Azadegan. Threading secure coding principles and risk analysis into the undergraduate computer science and information systems curriculum. In *Proceedings of the 3rd annual conference on Information security curriculum development*, pages 24–29, 2006.

[55] 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.

[56] David A. Wheeler. Flawfinder. https://dwheeler.com/flawfinder/.

[57] 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.

[58] 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.