Do CS Undergraduates Show Evidence of a Security Mindset without Formal Coursework? An Exploratory Qualitative Study

Michelle Jensen
mejensen5@wisc.edu
University of Wisconsin–Madison
Madison, WI, USA

Matthew Berland
mberland@wisc.edu
University of Wisconsin–Madison
Madison, WI, USA

Rahul Chatterjee
rahul.chatterjee@wisc.edu
University of Wisconsin–Madison
Madison, WI, USA

Abstract

As computer security threats continue to grow, it is critical that all computer science (CS) students develop foundational security principles, including the complex but essential concept of a security mindset. However, much of the existing security education literature lacks grounding in the learning sciences, often portraying students as passive recipients of facts rather than active co-constructors of knowledge. To address this gap, we conducted a qualitative study to examine evidence of a security mindset in situ, laying the foundation for future research on when and how this mindset emerges. We analyzed think-aloud coding sessions modeled on the Build-It phase of the Build-It, Break-It, Fix-It (BIBIFI) competition. Despite limited or no prior exposure to computer security, participants exhibited core aspects of a security mindset, including secure design practices and threat perception. These findings suggest that students can demonstrate meaningful security reasoning even without formal coursework, highlighting opportunities for low-overhead interventions to cultivate a security mindset. Our results inform future research and pedagogical design targeting foundational security thinking in undergraduate CS education.

CCS Concepts

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

Keywords

security mindset, computer security education, computer science education, post-secondary education, qualitative methods, constructivism

ACM Reference Format:
Michelle Jensen, Matthew Berland, and Rahul Chatterjee. 2025. Do CS Undergraduates Show Evidence of a Security Mindset without Formal Coursework? An Exploratory Qualitative Study. In *ACM Conference on International Computing Education Research V.1 (ICER 2025 Vol. 1), August 3–6, 2025, Charlottesville, VA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3702652.3744226

1 Introduction

The increasing prevalence of cyberattacks [23] and the use of our digital technologies in every aspect of society underscore the urgency of integrating security into undergraduate education. Yet, computer security remains an afterthought in many curricula. Although the Joint Task Force on Computing Curricula incorporated security into undergraduate CS standards in 2013 [17], many students still graduate without meaningful exposure to the topic due to limited course offerings, non-mandatory electives, and high prerequisite barriers [3, 9]. This lack of exposure is concerning given that undergraduates are expected to design and maintain complex software systems. Security should be foundational, enabling them to conduct threat modeling, apply best practices, and incorporate secure design principles.

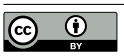
Research on how undergraduates learn computer security is still emerging. Much of the current work focuses on evaluating student performance (e.g., fact recall or error rates) or assessing specific tools and modules. However, performance metrics do not necessarily reflect conceptual understanding. To address this, we adopt a constructivist lens [10], emphasizing that students actively build knowledge through experience and prior learning. Applying this framework to security education offers a path toward deeper understanding, better preparing students to anticipate and defend against evolving threats.

This study is part of a broader initiative to equip all CS undergraduates with the practices, skills, and mindset necessary to recognize and mitigate security threats—without significantly burdening already full curricula. As a first step, we explore the following research question:

How do CS undergraduates, regardless of experience, exhibit instances of a security mindset while completing a security-focused BIBIFI task?

We answer this question through qualitative analysis of individual think-aloud sessions with CS undergraduates that mirror the Build-It phase of Build-It, Break-It, Fix-It (BIBIFI) competitions. Participants were tasked with reinforcing a simple game to prevent players from cheating, primarily through writing new code responsible for managing the game's state and for designing a passcode system. This activity was chosen to help make the concept of a threat tangible as a scaffold to help them engage with deeper security questions, many for the first time.

We conducted this study with fifteen undergraduate computer science students. We analyze their *think-aloud* sessions and the code they wrote to identify behaviors that reflect traits of a security mindset, particularly through the notions of *secure design* and *threat perception*. For example, some students discussed developing passcode systems to prevent cheating but also acknowledged the challenge of protecting passcodes from "cheaters" who could read the code. Only two of our participants had taken a formal security-related course.



ICER 2025 Vol. 1, Charlottesville, VA, USA

ACM ISBN 979-8-4007-1340-8/2025/08
https://doi.org/10.1145/3702652.3744226

The security mindset demonstrated by these students was often incomplete and inconsistent. However, the key takeaway from our study is that students do possess the foundational elements of a security mindset, which can be further developed through low-overhead interventions in CS undergraduate courses—such as a single BIBIFI assignment and a few short lectures on the fundamentals of computer security. We hope that our research will inspire further studies in security education that leverage the Zone of Proximal Development [48] and constructionism [26] for undergraduate computer science students.

2 Related Work & Background

One of the foundational principles in computer security is the *security mindset*. Although there is no universally accepted definition, we adopt a broad interpretation: the security mindset involves actively considering potential threats and taking steps to monitor or defend against them [8, 35, 37].

Schoenmakers et al. [34] further conceptualize the security mindset through three interconnected practices: (1) monitoring for vulnerabilities or attacks, (2) evaluating the feasibility of those threats, and (3) assessing their potential impact or damage. These components do not follow a strict sequence but rather function as a dynamic and reinforcing process. Embracing a security mindset thus requires a shift in how software is designed and developed—foregrounding adversarial thinking alongside functionality.

2.1 Undergraduate Security Education

Integrating computer security topics across the curriculum can yield significant benefits [5, 7, 38, 41]. Given that security applies to all CS topics, distributing it across the curriculum not only allows us to discuss topic-specific security concerns when relevant (i.e. SQL injections in a database course) but allows us to introduce concepts early and reinforce them often. Approaches to incorporate these concepts into non-security courses include examples such as security-based assignments [21, 30], writing attack scripts for their own code [19], and security review interventions [39]. Some of these approaches were used for students at the CS0 or CS1 level. To offset the potential additional work for instructors to incorporate security concepts into their courses, such as assignment development, prior work has developed and evaluated interactive security modules at various levels [11, 12, 40–42] and have also developed cybersecurity concept inventories [22, 24, 29].

Other researchers have focused on evaluating security in common "traditional" CS learning resources, finding insecure code examples in textbooks [3, 43], undergraduate lectures and materials [2], and the popular programming Q&A site, *StackOverflow* [1]. In non-security courses, researchers have evaluated students' ability to code securely and found insecure code written by students. They also examined that awareness (or lack thereof) of specific topics in security affected their ability to write "clean" code in both embedded programming [15] and systems [2, 4] courses. Another group of researchers has developed a measure for students' self-efficacy of secure programming [6].

Previous work has also examined students who have taken or are currently taking a security course and uncovered common misconceptions (e.g., over-generalizations, incorrect assumptions, and conflation of concepts) [44].

Behind most of these approaches is an assumption that providing "good" tools and/or simply conveying the knowledge in class is sufficient for improving student understanding. The results of prior research predominantly comes from examining mistakes made by students or assessments of the students' ability to recall simple facts through pre- and post-tests. The latter is also noted as a trend in a review of the security education literature [47]. While the ability to quantify the errors that students make is helpful, we know little about *how* students learn various aspects of computer security. As such, further work is needed to understand not only their learning process but also what information and experiences we can draw upon to improve security education. In this work, we use methods from the Learning Sciences to investigate "security mindset" as one such means.

2.2 Constructivism & Learning

Constructivism is a wide-encompassing theory in the Learning Sciences, based on the work of Jean Piaget [16] and Lev Vygotsky [48], that people learn by constructing their own understandings of knowledge. In constructivism, learners are *active* participants in the process. Each individual brings their own set of knowledge, experiences, and pre-conceptions that form and shape the understandings that they construct. Learning cannot be a unidirectional "network transmission", nor are people "blank slates" when we encounter them; conveying information is an insufficient means to support understanding new content [10]. While there are several different approaches in constructivism (e.g., social, cultural, cognitive, constructionist), all agree that knowledge is constructed rather than transmitted. To our knowledge, there are relatively few examples of constructivist computer security education research. In contrast, general CS and programming education research is often strongly based on and commonly leverages constructivism and constructionism, such as Papert's LOGO [26], Scratch [20], teaching CS through the creation of video games [18], or using e-textiles (e.g., [13]). Constructivist theories of learning promote social, active learning contexts in which people "learn for deeper understanding" (as opposed to, say, memorization or rote skill acquisition). In the context of computer security, this would allow students to adapt to the ever-changing landscape of technologies and attacks.

2.3 Build-It, Break-It, Fix-It (BIBIFI) Activities

Build-It, Break-It, Fix-It (BIBIFI) security competitions were proposed in order to help understand the tools, languages, processes, and methods that could lead to secure software as "a new software security contest that is a fun experience for contestants, as well as an experimental testbed by which researchers can gather empirical evidence about secure development" [33]. BIBIFI activities, from their inception, are inherently security-focused and have not been widely used in non-security-based contexts. In contrast to the more popular Capture the Flag (CTF) competitions, which traditionally focus on practicing from an attacker's perspective, BIBIFI activities incorporate defending and fixing components. This more closely mimics the real-world security "workflow" where a

piece of software is (ideally) designed securely, then once an attack is launched or a vulnerability discovered, team works towards implementing a fix — encapsulating Schneier's idea of computer security being a "constantly evolving arms race between attacker and defender" [35]. Competitions occur in three sequential phases where competing teams: (1) create a piece of software according to some specifications, (2) find and document bugs or vulnerabilities in others' implementations, and (3) attempt to fix any discovered issues in their implementation. Teams are scored in two different categories, with "security-relevant" matters being weighted more in the scoring.

Research using and revolving around BIBIFI competitions is still relatively immature, and there is much to learn about them given their potential. The existing literature focuses on categorizing the mistakes made or existing vulnerabilities made by senior software engineers [46] and students in a security class [14]. BIBIFI has also been seen used in several security-focused classes [28, 32, 45]. However, we know little about the actual discourse of students engaging with it or its impact, positive or negative, regarding learning when used in a formal education setting.

3 Method: Think-Aloud Build-It Study

The goal of the study is to elicit explicit conversations about organic security threats, observe how students assess the applicability and likelihood of different threats, and explore how they conceptualize solutions to protect their software. Constructivist assessment — how to know what people understand — is commonly used for theory building. This paper builds on work such as Sherin [36] exploring how to build better learning environments by assessing students' initial understanding of the topic. Thus, we are interested in understanding how students think about and approach software security, even in the absence of formal instruction on the topic; in the study, we are not focused on the specific solutions they devise. Gaining this understanding can help inform the development of effective interventions in undergraduate CS curricula to foster a security mindset without requiring drastic changes.

3.1 Build-It Activity Design

We model the task based on the "Build-It" phase of a BIBIFI competition [27]. Participants were asked to secure a linear, text-based adventure game, in which players progress through levels collecting items, finding health upgrades, and fighting enemies. As players progress, their game state (e.g., current level, health, health upgrades, and collected items) is updated. When a player chooses to pause, the game generates a passcode representing their progress, allowing them to resume it later.

The participants were provided with stub code to begin working on a specific task: developing the secure passcode system to enable game un/pausing and then secure other aspects of game.[1] We designed that tasks such that students who have completed CS2 (*which covers object-oriented programming and data structures*) would be adequately prepared to engage in the study.

Participants were explicitly informed that players of the game would attempt to cheat and that other participants might try to "break" their code (*Break-It* phase) — this statement framed the existence of an attacker within the context of the task. Beyond this, the threat was deliberately left open-ended without any detailed model. The game and its starter code is designed to offer multiple pathways for participants to implement security measures and complex enough that no participant could be expected to completely "solve" the task. Anticipated pathways included, but were not limited to: (1) encoding the passcode to obscure its contents and prevent cheating, (2) patching an exploit in the pause/resume feature that allowed infinite health upgrades, (3) ensuring the player's current health could not exceed the maximum limit when manually restored, and (4) enforcing boundary conditions on passcode data to maintain game integrity (e.g., preventing invalid health values or items beyond their intended availability).

By designing the task with these flexibilities, we allowed participants to approach security in ways that reflected their own reasoning and understanding, so that the data could yield insights into their preexisting security mindset.

Rationale behind the task choice. BIBIFI activities allow participants to engage in a software development task ("Build-It" phase) while considering realistic threat of security attacks ("Break-It" phase). Since this activity was likely their first encounter with computer security, making the existence of an attacker more tangible, it served as a scaffold (from Vygotsky's *Zone of Proximal Development (ZPD)* [48]) through simulation [31]. We designed this study to help participants recognize the presence of an attacker while keeping the security threat open-ended, preventing restriction to a single predefined attack type and encouraging independent realizations and responses to potential security threats.

Second, this task is simple, engaging — most students play games and are therefore likely familiar with game save codes — yet very hard to solve "correctly", even with security and cryptography training. This design constraint was intentional so that students could not find a "standard solution". Instead, the task was structured to encourage "open-ended" thinking and exploration through multiple potential solutions to find their limitations — in some sense, it was a small-scale, simultaneous building-, breaking-, and fixing-phase task. This approach was designed to make manifest the different ways that students conceptualize and perceive security threats.

Without sufficient scaffolding, students might not naturally *consider* security aspects in their development process. Our goal was to examine students' perceptions and intuitive understanding of security threats rather than test their formal cybersecurity knowledge. Similar to Young and Krishnamurthi's *adversarial thinking* [49], we did not expect students to anticipate or address all possible security threats comprehensively, but observe how they perceive different threats and adjust their software design.

Study protocol. The study is conducted over Zoom in 1-hour individual think-aloud sessions with a singular researcher. The participants are given a security-based prompt and a starter code sent at the start of the think-aloud session. The participants download and open the code using their choice of IDE/text editor to read and make modifications. We recorded the video of their computer screen and the session's audio. We transcribed the audio and took detailed notes of the screen recordings for further analysis. At the end of the session, we collected a copy of their code and a conducted

[1] More context on passcode systems and the specifics of the prompt can be found in Appendices A and B respectively.

a brief semi-structured interview to follow-up on their approach and thinking. Additionally, participants filled out a short survey which gathered their completed CS coursework (including security courses), program year, and other prior security experience. The protocol was reviewed by the Institutional Review Board (IRB) of our university and approved as a "minimal-risk human subject study".

Participants were allowed to utilize online resources (e.g., websites, online tutorials, documentation, notes from classes). Since LLMs could be used to list potential attack and defense suggestions, participants were prohibited from using LLMs such as Copilot or ChatGPT. The think-aloud coding sessions enabled us to collect *live* indicators of a security mindset, allowing insight into possible avenues and completed successful attempts and failures to account for security threats.

3.2 Participant Recruitment & Data Collection

We recruited 15 students from an R1 university in the Midwestern USA to participate in our study through a department mailing list and flyers. Interested students submitted a screening survey that asks if their major (intended or declared) was CS and if they had completed a CS2 (or equivalent) course. Those who met this criteria were then invited to participate in the study.

Thirteen of the participants were male, one was female, and one preferred not to answer. The participants were all at various points in their undergraduate careers. Five participants were sophomores (2nd year), seven juniors (3rd year), two seniors (4th year), and one was in their 5th year. In addition to the CS major, six participants had secondary majors: Data Science (DS) (3), Mathematics (Math) (2), Computer Engineering (CE) (2), Legal Studies (LS) (1), and Psychology (PSYCH) (1). Participants reported a range of CS coursework beyond an introductory programming and foundational computing classes as displayed in Fig. 1. Two participants (P8 and P15) cited previous security experience, having taken a cryptography class.

P	Yr.	Gender	Major(s)	Advanced CS Courses Taken
P1	3rd	Not Given	CS, DS	AI, BD
P2	2nd	Male	CS, Math	–
P3	3rd	Male	CS	–
P4	3rd	Male	CS	ALGO, DSPROG
P5	4th	Male	CS, DS	ALGO, DSPROG, OS, AI, BD, HCI
P6	3rd	Male	CS	ALGO, DB, AI
P7	3rd	Male	CS, CE	–
P8	5th	Male	CS	ALGO, DB, OS, CRYPTO, OPT
P9	2nd	Male	CS	–
P10	3rd	Female	CS, DS	ALGO, OS, AI
P11	2nd	Male	CS	AI, BD
P12	4th	Male	CS	ALGO, NM, PL, GRAPH
P13	3rd	Male	CS, CE	–
P14	2nd	Male	CS, LS, PSYCH	ALGO, AI
P15	2nd	Male	CS, Math	ALGO, AI, CRYPTO, PL

Figure 1: Demographics of the participants and the advanced CS courses they have taken. All students have taken Object-Oriented Programming & Basic Data Structures (OOP), Advanced Data Structures & Programming (ADSP), and Intro to Computer Engineering (INTROCE), and most have taken Discrete Math (DM) and Machine Organization (MO). Some students have also taken courses like: Data Science Programming (DSPROG), Artificial Intelligence (AI), Algorithms (ALGO), Big Data (BD), Databases (DB), Cryptography (CRYPTO), Computer Graphics (GRAPH), Operating Systems (OS), Human Computer Interaction (HCI), Numerical Methods (NM), Optimization (OPT), Programming Languages & Compilers (PL).

3.3 Qualitative Data Analysis

Our research question focuses on exploring how and when CS undergraduates, regardless of experience, demonstrate a security mindset or threat modeling. *This is explicitly distinct from the security of the code that they write.* In our analysis, we do not evaluate gaps or flaws in their thinking, such as if their approach introduces more vulnerabilities or if there is a "more secure" solution. As a result, we utilize the general overarching definition of *security mindset* as a basis for our qualitative coding scheme's initial pass. After constructivist work on learning for understanding by Sherin [36], we base our coding scheme on both an *a priori* target content breakdown and a more empirical catalog of the ways that students perceived their actions. As such, we use an inductive approach to coding. We began with an initial pass of coding in which instances of security related thinking were marked and labeled with a summary description. Researchers then met and discussed the initial labeling on a sample of three Sessions, examined the individual labels for similarities, abstracting one level of specificity. For example, *use SHA-256 for the passcode* and *apply a Caesar cipher to passcode* both involve applying cryptographic concepts, and thus were merged into the *Cryptographic Techniques* code. These preliminary codes were iteratively revised, refined through group discussion until we reached agreement on two high level codes and then reapplied to the data until they reached a stasis. One researcher coded the data in close discussion with the whole research group.

In our data, we see two sub-elements of a security mindset quite clearly: *threat perception* and *secure design*, as elements of the "Protect & Defend" roles and "Design & Develop" roles, respectively [34]. We also see various ways that the relationships between monitoring, evaluating, and investigating are described by Schoenmakers et. al. [34]. Two highest-level codes were aggregated from several sub-codes that appeared as we analyzed the sessions and code for instances of them securing a part of the game or discussing how to secure the game or prevent an attacker. Overall, *threat perception* is evidenced by participant discussing potential attacks and threats, hypothetical strategies to counteract those threats, and acknowledgment of the insecurity of an implementation. *Secure design* is evidenced by good coding practices (in accounting for possible threats) and attempting to design a secure passcode system. The complete set of high-level codes that we use to describe the data is shown in Fig. 2.

4 Results

We discovered two ways students exhibit a security mindset: (1) through identifying realistic threats that their software might be exposed to, which we call *threat perception*, and (2) make design choices to overcome such threats, which we call *secure design*.

Code	Definition	#
Secure Design	Describing a design decision with the *intent* to enhance security.	**13**
– Encoding of Passcodes	Prevent cheating by not leaving the game state information in plaintext.	11
– Cryptographic Techniques	Hides the game state information in the passcode using encryption or hashing.	5
– Stenographic Techniques	Obscures the game state information in the passcode.	7
– Defensive Tactics	Prevent cheating by enforcing rules or other attacks.	9
– Enforce Rules of Game	Ensure specific rules in the game is being followed.	7
– Misc. Defensive Tactics	Secure the game from other types of violations/attacks, e.g., limit on how many wrong passcodes can be entered	7
Threat Perception	Identifies potential attacks or attackers for the given task.	**13**
– Perform an Attack	Performs an attack on the stub code or in their code/design.	8
– Describe potential attack or attacker	Describes an attack on the game or identifies an attacker.	9
– Acknowledge insecurity of approach	Points out a flaw in the stub code or in their code/design.	7
– Recognize task is security-related	Explicitly mentions that the goal is to secure the game.	6

Figure 2: The qualitative codes and their definitions applied to the data as a part of the analysis. Sub-codes are indicated by indentation.

4.1 Threat Perception

We define threat perception as the understanding of realistic threat that their software might be subject to. The threat perception is a requirement towards security mindset. To prompt participants' threat perception, we framed the problem of potential threats and attacks as "players will attempt to cheat" without outlining possible skills or resources that may be necessary to conduct those attacks. Nevertheless, we observed participants mentioned or demonstrated several instances of threat perception during the think-aloud sessions. We identified any design decisions or code snippets that are potentially related to security during our initial coding, which we then refine and categorize as threat perception if it is related to one the following four: (1) describe a potential attack; (2) attempt to perform an attack (on a base implementation or on their own implementation); (3) acknowledge that some aspect of an implementation is not secure; or (4) recognizing the security aspect of the task from the prompt. We did not expect participants to model all possible threats, nor evaluate the feasibility of an attack or type of attacker existing. Some level of threat perception was demonstrated by all but two participants — P10 and P11. P10 and P11 did not even consider the problem to be security related and designed their solution assuming no possible threat exists (which is clearly insecure).

Describing Potential Attacks. Out of the 13 participants who demonstrated threat perception, nine (P4, P6–P9, P12–P15) of them described (or attempted to describe) at least one potential attack or attacker. P14 described that a cheating player with high technical skills could somehow obtain a copy of the game code and reverse engineer it to find ways to circumvent their defenses and understand the passcode encoding. P14 also mentioned integer under/overflow by inputting negative values and using a forged passcode to get more health upgrades than the limit or to skip to a level without having the items required to be previous levels. Both P12 and P14 noted attacks with a forged passcode where the current health could exceed the maximum allowed health. Two participants (P9 and P12) pointed out an unlimited health upgrade cheat that could occur from exploiting the resume and level map generation features in the game's design. Thus they demonstrated security thinking and identifying security flaws in the software's design.

P4, P6, and P15 all explained attacks on their passcode design. They mentioned that a player could crack the scheme by either brute-forcing all possible combinations or generating and testing enough passcodes to "detect a pattern." Finally, and most commonly, attacks that involved a cheating player simply reading and interpreting a plaintext passcode and thus letting them create passcodes that they never obtained from the game. This was described by P7, P8, P12, P13, and P14.

One interesting case of this happening was with P13, who started to consider the real-world example we mentioned in our prompt first to help them frame the attack and its defense:

> ***P13:*** *"How did Metroid do it?"*
> ***Researcher:*** *"Why are you thinking about Metroid right now? How does that relate to the task you are trying to accomplish?"*
> ***P13:*** *"I was thinking if they [the developers] did really check for progress, or could the player have cheated their way in? I was assuming that Metroid checked for progress, but I'm not sure because I've never played the game."*

While information about the design of *Metroid*'s (a video game from the 80s) passcode system is easily available on the web, P13 did not investigate further. As a result, they made an (incorrect) assumption about the security of *Metroid's* passcode system.[2] Nevertheless, framing their mindset in such a way was a strategy they employed to conceptualize a way one could try to cheat and thus start thinking about countermeasures to that attack by example.

Performing An Attack. Eight participants (P1, P2, P4, P5, P7, P8, P12, P13) attempted to perform an attack on either the insecure stub implementation or their implementation. After reading the prompt and playing the game with the base implementation a bit, P4 started to attack the provided starter code after reviewing the section of the prompt that mentioned cheating players:

> *"Then I'll just check if I can cheat the game [the base implementation]. What are the level names?" [reviews the provided list of level names] "Okay, I'm gonna see if I can directly skip to 'Ruin of the Ancients'." [provides*

[2] *Metroid*'s password system never checks it for a logical game progression. Rather, the game checks for a "valid" password using a checksum. Checksums were a pretty common practice in game password systems.

> *the game with "Ruin of the Ancients" as the passcode; the game skips resumes at a level they had not reached before in gameplay] "We can just comment that the passcode is just level name. Very easy to cheat."*
>
> *–P4*

Their attack took advantage of the fact that the stub implementation's passcode was simply the name of the current level (in plaintext) and, as such, could be used to skip to any level, as demonstrated by the above quote. This suggests that attempting and succeeding at the attack helped them consider ideas to strengthen their implementation. P13 also conducted the same attack. The most common attack performed by participants (P1, P2, P4, P5, P8, P12, P13) happened while they played the game to get familiar with it. They exploited the fact that the health restore action allowed players to input any number. Using this "hack", they could exceed maximum allowed health.

Acknowledge An Implementation's (In)Security. The third way participants demonstrated threat perception was by verbally acknowledging the insecurity of an implementation, whether their own or the provided stub implementation. These are situations where participants noted that something was "easy to cheat" or "insecure" and were able to follow-up with why they thought so. We did not include situations indicated by protestations of low self-confidence in their abilities. Instances of these acknowledgments occurred for seven participants (P3, P4, P6, P7, P12, P13, and P14). For example, here, P6 talks about how their implementation adds some level of security through the use of a Caesar cipher but likely not as much as other cryptographic methods:

> *"I guess it doesn't really offer too much additional security given the Caesar cipher is kinda like, I don't know, pretty basic? I'm guessing it's completely useless by most cryptographic standards but... I don't know. I assumed it might help a bit more [than nothing]." –P6*

Besides noting their own implementation's passcode, participants pointed out that the stub implementation's plaintext passcode was insecure. Some were even surprised that it was simple to crack. Another example of this subcategory comes from P13:

> *" I don't think I've been able to implement a way to make sure that they [the player] really have completed the level. Probably the one layer of security that I've tried to implement is adding that cipher to the passcode." –P13*

They are aware of an attack but know they have not addressed it due to their implementation only addressing a different attack.

Is Task Security Related? Lastly, we took note of whether a participant framed the task provided as "anti-cheat" or "securing code", as this was one major step to start thinking about threats. Students were explicitly asked to frame the task in their own words after reading the prompt aloud. The framing of the task to involve security was expected to be universal given the scaffolding we provided in the task's design; however, roughly two-thirds of the students did not (P1, P2, P5, P8, P9, P10, P11, P14, P15) and the remaining six did (P3, P4, P6, P7, P12, P13). P11 explicitly denied the existence of a potential attack while working through an initial approach to store all GameProgress objects[3] in an ArrayList.

> *"For simplicity [regarding my implementation], I know there are no crazy people who will start up like a million GameProgresses."–P11*

During their discourse around an `ArrayList` possibly reaching capacity, they never mentioned the impracticability that could deter a cheating player from using that approach. Rather they decided no one would think to do that. As a result of this difference in framing with regards to the purpose of the task, the participants divided themselves neatly into two groups.

Summary of students' threat perception. Most participants identified attacks that involved decoding the passcode and its information. Once the cheating player has done that, they can start creating and using forged passcodes to (1) skip to later levels, (2) get "free" items and health upgrades, or (3) break other rules in the game. Some other threats discussed revolved around other features of the game. With these threats in mind, participants used them to inform their work in designing defenses. We elaborate on those defenses in the next section.

4.2 Secure Design

The second category of evidence of a security mindset that was observed is instances of secure design. Any design decisions implemented or discussed that participants explicitly explained could improve the security of the passcode system or game overall fell into this category.

As discussed in Section 3.1, we are not concerned about actual security issues that could result in implementing code (e.g., incomplete functionality, implementation mistakes, insecure coding practices such as resource leaks). Our analysis is concerned with discovering evidence of secure design, not the quality of the design decisions. A total of thirteen of the participants demonstrated evidence of secure design.

Encoding the Passcode. To give participants a context about passcodes and how game state could be saved, the study prompt contained an implicit design suggestion that encoding was a part of the passcode systems in older video games. It never pointed out that this could be a strategy to defer cheating players or as a requirement. Participants had the freedom to use or ignore this suggestion. Among our participants, 11 designed a technique for generating and parsing passcodes without leaving the passcode's game information in plaintext. In terms of encoding the data, the eleven fell into one of two categories: traditional cryptographic methods or stenographic techniques.

Five participants used traditional cryptographic approaches like encryption and hashing to encode the passcode. Three students (P2, P6, and P13) used shift ciphers.[4] Although shift ciphers are vulnerable to simple known plaintext attack — knowing one pair of plaintext and ciphertext can reveal the key (shift value) — none of them had prior formal security or cryptography experiences.

[3] An object responsible for storing and maintaining aspects of the game state.

[4] Shift ciphers are an encryption method where each alphanumeric character in the text is "shifted" to another character by some fixed amount. Ex. "`Hello World`" with shift of 5 would become "`Mjqqt Btwqi`".

```
public String generatePasscode(){
    //Convert the current and max health values from decimal to
          hexadecimal
    String currentHealthBytes =
          Integer.toHexString(getCurrentHealth());
    String maxHealthBytes = Integer.toHexString(getMaxHealth());
    boolean[] itemTable = new boolean[9];
    String itemsCollected = "";

    // Marks element at index (levelNum-1) true if the player has
          the item found in the corresponding level
    for(String item : currentItems) {
        switch(item) {
            case "Lucky Charm":
                itemTable[0] = true;
                break;
            case "Ascendant Boots":
                itemTable[1] = true;
                break;
            // switch cases continue for each of the major items
        }
    }
    // Builds a string of all 9 items in order with a "1" if item
          obtained and "0" otherwise. Ex. Having the items of levels
          1 & 3 would result in the String "101000000"
    for(int i=0; i<itemTable.length; i++){
        if(itemTable[i] == true) itemsCollected += "1";
        else itemsCollected += "0";
    }
    return currentHealthBytes + maxHealthBytes + itemsCollected;
}
```

Figure 3: P12's code which combines the use of hexadecimal and binary to encode the game information. Comments added by us for readability.

Some students had seen a Caesar cipher, a specific type of shift cipher, tangentially in prior coursework (e.g., Harvard's CS50X course). In other cases, participants did not know the technical term(s) for their approach, but it was a solution they reasoned out using their current set of problem-solving skills. In their verbal thinking, participants explained that the issue they were trying to solve involved "hiding the values" so that no one could easily interpret the passcode. P14 briefly also considered using SHA-256 (Secure Hash Algorithm 256) to "scramble" the item names in the passcode before continuing to opt for a different approach. P8 used Advanced Encryption Standard (AES). AES is a cryptographically strong cipher; given that they were one of the participants to have taken a cryptography class, it explains why they knew about it and opted to use a more complex method of hiding the passcode information. While P15 had cryptography experience, they opted not to use it. Their reasoning was two-fold. First, the possible number values that the passcode would contain, such as health upgrades, weren't all prime numbers and thus could not be used existing cryptographic schemes.[5] The other reason is that they said their approach was a sufficient defense.

Seven participants (P1, P3, P4, P7, P12, P14, P15) used steganographic techniques to encode the passcode. We observed three different techniques from these participants: data compression, change of base, or a custom dictionary. (Here, change of base is defined as any method to convert the text to a different numerical base.)

[5]Stronger cryptographic schemes often use prime numbers and algebraic group theory to make them impossible to crack in a reasonable amount of time.

Item Name	Output String
Lucky Charm	`Q2$]`
Ascendant Boots	`O9/{`
Abyssal Rope	`A7;-`
Temporal Compass	`U8&\`

Figure 4: An example of a portion of the dictionary created by P4 to encode the item names.

After concatenating the game data into a string, P1 encoded that string into Base64 and compressed it using Java's Deflater class before outputting it to the terminal. P1 compressed the final passcode to improve the usability of copying and typing the passcode.

P12 used a boolean array to denote which items have been collected and obfuscating decimal number values, such as current health, as hexadecimal values. (A snippet of P12's code is given in Fig. 3). P3, P4, P7, P14, and P15 created mappings between values and manually selected strings; this is what we call use of custom dictionary approach. Here the dictionary acts as the key for a substitution cipher to obfuscate the content of the passcode. Some chose words by association, others pick sets of "random looking" same length strings, while another had a tool generate a set of words. An example of one of these can be found in Fig. 4, which shows the key pairs P4 created and then stored into a hash table. Almost all custom dictionaries were static, meaning the values always mapped to the same "random" string. An exception was P15, who had a different mapping strategy for their current health values than items and level to "make it trickier for the player to figure out the pattern." They opted to use the current number of health upgrades when deciding which word from the table to use for the current health value. For example, if the player's current health was 5, and they had 3 upgrades, the word in the passcode would be "empower," but having the same health and 4 upgrades would result in the word "cockpit." Having a current health of 4 and 4 upgrades would also result in the word being "empower."

Our participants employed various cryptographic and steganographic techniques — though insecure — to protect the passcode. However, they struggled to articulate their approach in terms of security properties. This suggests that BIBIFI activities could help instill a security mindset by introducing fundamental concepts like secrecy and integrity.

Four participants did not try to hide the passcode. P5 and P9 did not engage with the passcode system during their sessions; rather, they opted to spend it investigating and incorporating other defenses, which we elaborate upon in the following section. While the last two participants ignored the implicit suggestion about passcode encoding, leaving the game state information in plaintext. They did not discuss or attempt to hide information in their passcode systems, with P10 simply concatenating the information together with delimiters and P11 writing the information directly to a text file that had no restrictions on file access.

Additional Defensive Tactics. Besides protecting the passcode, a number of student participants used various techniques to secure their code or the game logic, which we refer to as *defensive tactics*.

Any defense that was not related to hiding the game data in the passcode belonged to this category. Nine participants (P2, P3, P4, P5, P7, P9, P12, P13, P14) demonstrated at least one defensive tactic.

The most frequent of these tactics was to enforce explicit and implicit game rules. This was demonstrated by P3, P4, P7, P9, P12, P13, and P14. Design decisions about the overall game, such as the level progression requirements and number of health upgrades per level, were given in the prompt to provide enough context of the game to work with. The prompt never mentioned that they were required to enforce them as rules, but we anticipated this as an avenue to explore when designing defenses. P7, P9, P12, and P14 discussed or added a check to ensure that a user-inputted passcode contained all the required items from the previous levels to prevent them from skipping to levels without the correct items. For example, if the player entered a passcode stating they were at Level 3 (The Cliffs of Insanity), the passcode also needed to state they had the "Lucky Charm" and "Ascendant Boots" items, requirements to reach that level. Five participants (P3, P7, P9, P12, P14) enforced an implicit rule that limited the number of health upgrades a player could have. Each level only had 2 optional health upgrades, indicating that there is a maximum to what the player's max health is, assuming regular play. P7 in particular picked up on enforcing this rule quickly, stating the following almost immediately after reading the prompt and being provided other instructions:

> *"So right off the bat, the first thing that kind of makes me think about is that every level has two optional health upgrades. They're optional. So, for example, if someone is on the third level, they should only have 4 max. They can have anything below that because if they have six or maybe eight, then that is kind of a red flag." — P7*

The most common rule enforcement was making sure the player's current health did not exceed the maximum. This was demonstrated in six participants: P3, P5, P7, P12, P13, and P14.

The remaining evidence of defensive tactics was varied enough that some of them were only seen in one participant and thus were coalesced into a *Miscellaneous Defensive Tactics* category. P7 explicitly chose to make sure to only relay user-relevant feedback when there was an error to prevent an attacker from gaining technical information (i.e., displaying a stack trace from an exception). Two participants, P4 and P14, discussed adding junk information to their passcodes to sidetrack a cheating player's attempt to figure out the passcode system. P4 also communicated their desire to prevent cheating players by locking them out of the passcode system after a few failed attempts and keeping a record of previously generated passcodes to verify against. In a similar vein to the latter, P13 wanted to log a player's game activity to verify entered passcodes and reject any that did not match. Other defenses considered by P14 include a verification check (at a high level) to detect forgeries, explicitly punishing an attacker when caught cheating (i.e., caught trying to give extra health will result in continuing the game with low health) or other annoyances to dissuade them, and preventing integer under/overflows by adding a check for inputted negative numbers. Participant 3 discussed how encapsulation and Java's access modifiers can prevent outside code, from an attacker, from accessing and manipulating data fields. Lastly, P2, P7, and P13 utilized random number generation as a part of their encoding to prevent an attacker from detecting a pattern.

5 Discussion

In this paper, we investigate how undergraduate computer science students demonstrate traits of a security mindset while completing a security-focused BIBIFI task. Our analysis (in Section 4) shows that students without formal security training can engage meaningfully with security problems and exhibit elements of a security mindset. However, their engagement is uneven and often unpredictable. We do not claim that these students are experts, nor that they would continue to exhibit these traits outside the structured environment we provided.

Notably, students independently attempted to solve security-related challenges, offering insights into how we might teach security thinking. We highlight instances where participants demonstrated secure design and threat perception intentionally, as well as cases where they implemented potentially security-relevant behaviors without explicitly stating security as a goal. For example, participant P1 included the line `System.out.println("Cannot read save")` to handle exceptions when decoding the passcode. This may reflect an intent to avoid revealing information to a potential attacker if a forged passcode is used—though P1 did not state this explicitly. Regardless of intent, this design choice also improved usability by clearly handling errors.

Across several examples, we observed participants unintentionally producing more secure designs, underscoring the potential for students to engage with security concepts even without formal training.

The intentional instances of secure design and threat modeling by participants (i.e., using cryptography, ensuring proper game progression) were typically specific to the context of the prompt about the game's passcode system. However, their responses varied across broader categories such as hiding information from an attacker, ensuring valid user behavior, and detecting & deterring attacks. While the prompt was considered to be cryptography-related, students tended toward non-cryptography-related approaches. P8 and P15's cryptography experience likely narrowed their thinking in terms of attacks and defenses due to their pre-existing experience with cryptography and cryptography-adjacent concepts.

As mentioned in Section 4.1, despite explicit prompting, some participants did not see the task as security-related, and, as a result, they engaged only minimally with the security content. On average, participants who **did** frame the task as being "security-related" had more instances of demonstrating both secure design and threat perception. This correlation aligns with the results of Schoenmakers et. al. [34] that different facets of a security mindset are interrelated and internally correlated. Several possible factors likely contributed to some of the participants minimally engaging with the security aspect of the activity, including the length of the session or its existence outside of a classroom context. The scaffolding provided was clearly inadequate for those participants, especially P10 and P11, though we had considered it manifest. This is a possible limitation of the study - if we are studying how students organically consider security threats when minimally prompted, then we will

find that some significant percentage of those participants will not organically consider security threats. Further investigation and intervention are needed to determine which features are more likely to prompt organic security-focused responses (without explicitly telling them something akin to "you are expected to address security threats") and thus could be utilized in the design of future tools. That said, our data in aggregate show that, with some prompting, many (if not most) students without prior formal computer security experience can demonstrate some aspects of a security mindset.

Perhaps the biggest takeaway is that courses should take advantage of the understandings that students are already showing. It is not a matter of learning something entirely new and maintaining that new understanding but instead of nurturing the seeds of existing understanding. This is consistent with our constructivist theoretical framework, which suggests that most learning is building on, reframing, and coming to use more standard language to describe existing deeper understandings rather than "depositing" entirely new knowledge in students.

We propose incorporating assignments (or additions to them) that provide an open-ended threat model to consider when completing the assignment. The threat model should be reasonable for the course content and level. Rather than giving a simple list of TODOs, students can engage more deeply with security thinking to inform their decisions. One such example of an assignment in action could be a programming class assignment that focuses on exception and error handling. Ideally, these kinds of assignments should be done consistently and frequently throughout the course and/or program and can be done so with minimal to no additional lecturing about security.

Future work can investigate good areas of curriculum and/or kinds of assignments to utilize this. Additionally, future work should investigate iterative processes (such as the Break-It, and Fix-It parts of BIBIFI), in which students can go back and consider new attacks and refine their approach. This could allow students' current thinking to mature. Furthermore, despite us seeing evidence of security thinking, it is unknown if certain types of security thinking are more prevalent in students naturally. For example, hypothetically, students might consider only security threats that have malicious intent or are motivated by financial gain. As a result, they largely neglect to consider threats with other motivations such as activism (hacktivists) or accidental damage. Ascertaining this could help inform where to focus interventions to help develop a well-rounded security mindset.

6 Conclusion

To understand if and how undergraduate CS students without prior security education could reason about security threats, we conducted an exploratory study that had students work on a security-focused task with minimal scaffolding. Our purpose in doing so is to enable further studies on a security mindset in all CS undergraduates. Analyzing the data from the think-aloud sessions, we saw that participants showed traits of a security mindset through secure design and threat perception. Some students did not engage in thinking about threats or did so minimally even with our scaffolding. Further investigation is needed to determine what factors have a greater chance of prompting security-focused responses. Despite this, most students can show signs of a security mindset, even if incomplete, provided minimal prompting.

Computer security is a subfield that many undergraduates perceive as "intimidating" and "difficult" [25] and therefore are reluctant to interact with. CS educators will hopefully be able to use the finding that, with the help of only minimal prompting, students will typically show a security mindset without prior formal experience. This finding can also, hopefully, empower students to explore computer security.

A Additional Context About Game Passcode (or Password) Systems

Due to limitations and restrictions revolving non-volatile memory (and subsequently internal and external memory) early video games, particularly on consoles, were unable to maintain save files. Games that wanted to allow players to be able to resume play from their current state at a later time implemented a password (also sometimes known as a passcode) system in which the idea was to "encode" information about the game's progress and then have the player input it at a later time. They were designed with the intent that you needed to have been given a passcode before and not skip past a part of the game the player didn't complete. Often on top of converting the bits into an alphabet for the player to enter the code, other measures were used to avoid players from cheating such as bit shifting, checksums, or a secret unique to each cartridge.

B Prompt

Before the use of game saves, often due to technical restrictions, video games used passcodes to allow players to pick up from where they left off in a game. The game provided a String of seemingly random characters that the player would need to write down and then they could enter it later and resume play. In actuality, this String encoded necessary game information.

One such example comes from the original *Metroid* game for the *Nintendo Entertainment System*. The passcode "`—— -m C3 -y000 00y03`" puts a player in the final area before the final boss with all possible items and upgrades.

Your task is to implement/edit the code for the passcode system to encode the data for a game called "Epic Escapades: A Textual Odyssey". You might want to start out by playing the game for a few minutes to get a feel for how the game works! Players should only be able to use a code if it was generated by the game itself. Note that players will try to "cheat" the game and should be prevented from doing so. You will want to focus (mainly) on the `GameProgress.java` file. Feel free to make any changes you feel necessary, EXCEPT for changing the existing method signatures. You can use any resources (e.g. documentation, tutorials) you would like as well except for LLMs such as ChatGPT, Copilot, or Claude.

Here are some details about the rules of the game that should be taken into account:

(1) Each level has 2 optional health upgrades. Each health upgrade adds 1 to the player's max health.
(2) Every level has 1 mandatory item in it that is required to beat it. Here is the list of levels and their mandatory item:
    - Level 1 - Beginner's Luck (Lucky Charm)
    - Level 2 - Onwards & Upwards (Ascendant Boots)

- Level 3 - The Cliffs of Insanity (Abyssal Rope)
- Level 4 - Lost in Time (Temporal Compass)
- Level 5 - Welcome to the Future (Holographic Projector)
- Level 6 - City of Lights (Electric Pulse Gloves)
- Level 7 - The Labyrinth (Enigma Key)
- Level 8 - Ruins of the Ancients (Ancient Scroll)
- Level 9 - The Final Frontier (Nova Core Crystal)

(3) When the player resumes the game using a passcode, they will always start at the beginning of the level.

Acknowledgments

We thank the reviewers of our paper for their insightful comments and suggestions. The research is supported in part by the Baldwin Wisconsin Idea Grant and the NSF award #2350165.

References

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. 2016. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *2016 IEEE Symposium on Security and Privacy (SP)*. 289–305. https://doi.org/10.1109/SP.2016.25

[2] Majed Almansoori, Jessica Lam, Elias Fang, Kieran Mulligan, Adalbert Gerald Soosai Raj, and Rahul Chatterjee. 2020. How Secure are our Computer Systems Courses?. In *Proceedings of the 2020 ACM Conference on International Computing Education Research (ICER '20)*. Association for Computing Machinery, New York, NY, USA, 271–281. https://doi.org/10.1145/3372782.3406266

[3] Majed Almansoori, Jessica Lam, Elias Fang, Adalbert Gerald Soosai Raj, and Rahul Chatterjee. 2021. Textbook Underflow: Insufficient Security Discussions in Textbooks Used for Computer Systems Courses. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 1212–1218. https://doi.org/10.1145/3408877.3432416

[4] Majed Almansoori, Jessica Lam, Elias Fang, Adalbert Gerald Soosai Raj, and Rahul Chatterjee. 2023. Towards Finding the Missing Pieces to Teach Secure Programming Skills to Students. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 973–979. https://doi.org/10.1145/3545945.3569730

[5] Matt Bishop. 2011. Teaching Security Stealthily. *IEEE Security & Privacy* 9, 2 (2011), 69–71. https://doi.org/10.1109/MSP.2011.43

[6] Matt Bishop, Ida Ngambeki, Shiven Mian, Jun Dai, and Phillip Nico. 2021. Measuring Self-efficacy in Secure Programming. In *Information Security Education for Cyber Resilience*, Lynette Drevin, Natalia Miloslavskaya, Wai Sze Leung, and Suné von Solms (Eds.). Springer International Publishing, Cham, 81–92.

[7] Jean R. S. Blair, Christa M. Chewar, Rajendra K. Raj, and Edward Sobiesk. 2020. Infusing Principles and Practices for Secure Computing Throughout an Undergraduate Computer Science Curriculum. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 82–88. https://doi.org/10.1145/3341525.3387426

[8] Edward Bonver and Michael Cohen. 2008. Developing and Retaining a Security Testing Mindset. *IEEE Security & Privacy* 6, 5 (Sep. 2008), 82–85. https://doi.org/10.1109/MSP.2008.115

[9] CloudPassage. 2016. CloudPassage study finds U.S. universities failing in cybersecurity education. https://www.globenewswire.com/news-release/2016/04/07/1312702/0/en/CloudPassage-Study-Finds-U-S-Universities-Failing-in-Cybersecurity-Education.html

[10] National Research Council. 2000. *How People Learn: Brain, Mind, Experience, and School: Expanded Edition*. The National Academies Press, Washington, DC. https://doi.org/10.17226/9853

[11] Chad M. Dewey and Chad Shaffer. 2016. Advances in information SEcurity EDucation. In *2016 IEEE International Conference on Electro Information Technology (EIT)*. 0133–0138. https://doi.org/10.1109/EIT.2016.7535227

[12] Wenliang Du. 2011. SEED: Hands-On Lab Exercises for Computer Security Education. *IEEE Security & Privacy* 9, 5 (2011), 70–73. https://doi.org/10.1109/MSP.2011.139

[13] Deborah Ann Fields, Yasmin Kafai, Tomoko Nakajima, Joanna Goode, and Jane Margolis. 2018. Putting Making into High School Computer Science Classrooms: Promoting Equity in Teaching and Learning with Electronic Textiles in Exploring Computer Science. *Equity & Excellence in Education* 51, 1 (2018), 21–35. https://doi.org/10.1080/10665684.2018.1436998

[14] Kelsey R. Fulton, Daniel Votipka, Desiree Abrokwa, Michelle L. Mazurek, Michael Hicks, and James Parker. 2022. Understanding the How and the Why: Exploring Secure Development Practices through a Course Competition. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1141–1155. https://doi.org/10.1145/3548606.3560569

[15] Tiago Espinha Gasiba, Samra Hodzic, Ulrike Lechner, and Maria Pinto-Albuquerque. 2021. Raising Security Awareness Using Cybersecurity Challenges in Embedded Programming Courses. In *2021 International Conference on Code Quality (ICCQ)*. 79–92. https://doi.org/10.1109/ICCQ51190.2021.9392965

[16] W Huitt and J Hummel. 2003. Piaget's theory of cognitive development. *Educational Psychology Interactive* (2003).

[17] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Association for Computing Machinery, New York, NY, USA.

[18] Yasmin B. Kafai. 1996. *Learning Design by Making Games: Children's Development of Design Strategies in the Creation of a Complex Computationtional Artifact*. Routledge, 71–96.

[19] Michael E. Locasto. 2009. Helping Students 0wn Their Own Code. *IEEE Security and Privacy* 7, 3 (2009), 53–56. https://doi.org/10.1109/MSP.2009.66

[20] MIT Media Lab. 2025. Scratch - Imagine, Program, Share. https://scratch.mit.edu/ Accessed: 08 June 2025.

[21] Kara Nance. 2009. Teach Them When They Aren't Looking: Introducing Security in CS1. *IEEE Security & Privacy* 7, 5 (2009), 53–55. https://doi.org/10.1109/MSP.2009.139

[22] Ida Ngambeki, Phillip Nico, Jun Dai, and Matthew Bishop. 2018. Concept Inventories in Cybersecurity Education: An Example from Secure Programming. In *2018 IEEE Frontiers in Education Conference (FIE)*. 1–5. https://doi.org/10.1109/FIE.2018.8658474

[23] U.S. Department of Homeland Security. 2024. https://www.dhs.gov/topics/cybersecurity

[24] Spencer Offenberger, Geoffrey L. Herman, Peter Peterson, Alan T Sherman, Enis Golaszewski, Travis Scheponik, and Linda Oliva. 2019. Initial Validation of the Cybersecurity Concept Inventory: Pilot Testing and Expert Review. In *2019 IEEE Frontiers in Education Conference (FIE)*. 1–9. https://doi.org/10.1109/FIE43999.2019.9028652

[25] Vidushi Ojha, Christopher Perdriau, Brent Lagesse, and Colleen M. Lewis. 2023. Computing Specializations: Perceptions of AI and Cybersecurity Among CS Students. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 966–972. https://doi.org/10.1145/3545945.3569782 event-place: <conf-loc>, <city>Toronto ON</city>, <country>Canada</country>, </conf-loc>.

[26] Seymour Papert. 1980. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc., USA.

[27] James Parker, Michael Hicks, Andrew Ruef, Michelle L. Mazurek, Dave Levin, Daniel Votipka, Piotr Mardziel, and Kelsey R. Fulton. 2020. Build It, Break It, Fix It: Contesting Secure Development. *ACM Trans. Priv. Secur.* 23, 2 (2020). https://doi.org/10.1145/3383773

[28] Bryan Parno. 2020. Project 1: Build it; break it; fix it. https://course.ece.cmu.edu/~ece732/s20/homework/bibifi/SPEC.html Publication Title: 18-335 / 18-732: Secure Software Systems.

[29] Seth Poulsen, Geoffrey L. Herman, Peter A. H. Peterson, Enis Golaszewski, Akshita Gorti, Linda Oliva, Travis Scheponik, and Alan T. Sherman. 2021. Psychometric Evaluation of the Cybersecurity Concept Inventory. *ACM Trans. Comput. Educ.* 22, 1, Article 6 (Oct. 2021), 18 pages. https://doi.org/10.1145/3451346

[30] Vahab Pournaghshband and Hassan Pournaghshband. 2021. Teaching Security Notions in Entry-Level Programming Courses. In *2021 IEEE International Conference on Engineering, Technology & Education (TALE)*. 997–1000. https://doi.org/10.1109/TALE52509.2021.9678545

[31] Maggie Renken, Melanie Peffer, Kathrin Otrel-Cass, Isabelle Girault, and Augusto Chiocarriello. 2016. *Simulations as scaffolds in science education*. Springer International Publishing: Imprint: Springer.

[32] Philip C. Ritchey. 2020. Build-it, break-it, fix-it Project. https://people.engr.tamu.edu/pcr/courses/csce489/spring20/project/bibifi_details.html Publication Title: CSCE 413/713: Software Security Project.

[33] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Atif Memon, Jandelyn Plane, and Piotr Mardziel. 2015. Build It Break It: Measuring and Comparing Development Security. In *8th Workshop on Cyber Security Experimentation and Test (CSET 15)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/cset15/workshop-program/presentation/ruef

[34] Koen Schoenmakers, Daniel Greene, Sarah Stutterheim, Herbert Lin, and Megan J Palmer. 2023. The security mindset: characteristics, development, and consequences. *Journal of Cybersecurity* 9, 1 (May 2023), tyad010. https://doi.org/10.1093/cybsec/tyad010

[35] Charles Severance. 2016. Bruce Schneier: The Security Mindset. *Computer* 49, 2 (Feb 2016), 7–8. https://doi.org/10.1109/MC.2016.38

[36] Bruce L. Sherin. 2001. How Students Understand Physics Equations. *Cognition and Instruction* 19, 4 (2001), 479–541. https://doi.org/10.1207/S1532690XCI1904_3

[37] Ambareen Siraj, Nigamanth Sridhar, John A. Drew Hamilton, Latifur Khan, Siddharth Kaza, Maanak Gupta, and Sudip Mittal. 2021. Is There a Security Mindset and Can It Be Taught?. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy (CODASPY '21)*. Association for Computing Machinery, New York, NY, USA, 335–336. https://doi.org/10.1145/3422337.3450358 event-place: Virtual Event, USA.

[38] Ambareen Siraj, Blair Taylor, Siddarth Kaza, and Sheikh Ghafoor. 2015. Integrating Security in the Computer Science Curriculum. *ACM Inroads* 6, 2 (2015), 77–81. https://doi.org/10.1145/2766457

[39] Madiha Tabassum, Stacey Watson, Bill Chu, and Heather Richter Lipford. 2018. Evaluating Two Methods for Integrating Secure Programming Education. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 390–395. https://doi.org/10.1145/3159450.3159511

[40] Cara Tang, Elizabeth K. Hawthorne, Blair Taylor, and Siddharth Kaza. 2013. Introducing Security and Responsible Coding into Introductory Computer Science Courses. *J. Comput. Sci. Coll.* 29, 1 (2013), 148–149.

[41] Blair Taylor and Shiva Azadegan. 2008. Moving beyond Security Tracks: Integrating Security in Cs0 and Cs1. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. Association for Computing Machinery, New York, NY, USA, 320–324. https://doi.org/10.1145/1352135.1352246

[42] Blair Taylor and Siddharth Kaza. 2016. Security Injections@Towson: Integrating Secure Coding into Introductory Computer Science Courses. *ACM Trans. Comput. Educ.* 16, 4 (June 2016). https://doi.org/10.1145/2897441

[43] Cynthia Taylor and Saheel Sakharkar. 2019. ');DROP TABLE Textbooks;–: An Argument for SQL Injection Coverage in Database Textbooks. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 191–197. https://doi.org/10.1145/3287324.3287429

[44] Julia D Thompson, Geoffrey L Herman, Travis Scheponik, Linda Oliva, Alan Sherman, Ennis Golaszewski, Dhananjay Phatak, and Kostantinos Patsourakos. 2018. Student misconceptions about cybersecurity concepts: Analysis of think-aloud interviews. *Journal of Cybersecurity Education, Research and Practice* 2018, 1 (2018), 5.

[45] Daniel Votipka, Kelsey Fulton, Mike Hicks, and Michelle Mazurek. 2020. CMSC 388n - build it, break it, fix it: Competing to secure software. https://www.cs.umd.edu/class/winter2020/cmsc388N/ Publication Title: CMSC 388n - build it, break it, fix it: Competing to secure software.

[46] Daniel Votipka, Kelsey R. Fulton, James Parker, Matthew Hou, Michelle L. Mazurek, and Michael Hicks. 2020. Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It. USENIX Association, 109–126. https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-understanding

[47] Valdemar Švábenský, Jan Vykopal, and Pavel Čeleda. 2020. 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* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 2–8. https://doi.org/10.1145/3328778.3366816

[48] L. S. Vygotsky, Michael Cole, Sally Stein, and Allan Sekula. 1978. *Mind in society: The development of Higher Psychological Processes*. Harvard University Press.

[49] Nick Young and Shriram Krishnamurthi. 2021. Early Post-Secondary Student Performance of Adversarial Thinking. In *Proceedings of the 17th ACM Conference on International Computing Education Research (ICER 2021)*. Association for Computing Machinery, New York, NY, USA, 213–224. https://doi.org/10.1145/3446871.3469743