BREAK-IT: Understanding Novice Approaches to an Attack-Challenge Task

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

With increasing reliance on computing systems and the growing frequency of cybersecurity incidents, it is important for CS undergraduates to develop foundational security skills before entering professional roles. In particular, students should be able to recognize and reason about potential security vulnerabilities in software. However, existing approaches to integrating security into the CS curriculum often emphasize narrow areas such as secure coding or highly technical topics like cryptography or software security, rather than fostering a broader perception of security threats. In this paper, we examine how undergraduates conceptualize and identify security threats by analyzing how they attempt to find "attacks" in other students' code. We conducted a think-aloud study with 15 CS undergraduates at a US-based R1 institution who had no formal training in computer security. Participants analyzed peer-developed text-based video game implementations to identify potential vulnerabilities, drawing on their prior experience implementing a similar game in an earlier "Build-It" task. Our analysis shows that students employed systematic, hypothesis-driven strategies, including unit testing, edge-case exploration, and controlled experimentation, while also drawing on prior experiences both inside and outside the classroom. Although most students attempted to validate whether an attack was successful, several stopped after identifying a single vulnerability, leaving additional issues unexplored. Based on these findings, we offer recommendations for CS instructors and curriculum committees on integrating foundational security concepts into programming assignments to help students better recognize and reason about computer security threats.

CCS Concepts

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

Keywords

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

ACM Reference Format:
Michelle Jensen, Matthew Berland, and Rahul Chatterjee. 2026. BREAK-IT: Understanding Novice Approaches to an Attack-Challenge Task. In *Proceedings of the 31st ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2026), July 10–15, 2026, Madrid, Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3803400.3809379

ITiCSE 2026, Madrid, Spain

ACM ISBN 979-8-4007-2634-7/2026/07
https://doi.org/10.1145/3803400.3809379

1 Introduction

The computer security landscape is rapidly evolving, with breaches, privacy violations, and other harms carrying clear social, financial, and safety consequences. Yet many undergraduate CS students face barriers to engaging with security, including perceived difficulty [23], extensive prerequisites, and limited course availability [3, 4, 8]. Consequently, students often graduate without exposure to basic security concepts, potentially leaving the workforce underprepared to recognize and address threats [2].

Prior efforts have focused on integrating specific security topics, such as secure coding, web vulnerabilities, cryptography, and software security [13, 20, 29]. While valuable, these approaches are often not sufficient on their own; students also need a general, transferable understanding of security — a *security mindset* with realistic threat perception. This need is emphasized in recent curriculum guidance: "...the CS2023 SEC knowledge area focuses on ensuring that students develop a *security mindset* so that they are prepared for the continual changes occurring in computing" [18].

In this context, a security mindset refers to the ability to perceive potential threats to a system, to reason about how systems might fail or be misused, and to systematically explore unintended or harmful software behaviors. Importantly, developing such a mindset does not necessarily require advanced or specialized security knowledge. Instead, it may build on skills students already practice in core CS courses, such as programming, debugging, and testing [36].

Introducing new standalone courses into an already crowded CS curriculum is challenging [20]. A key question, then, is how security-related thinking can be embedded into existing courses with minimal disruption. Addressing this question requires understanding how students currently reason about security-relevant problems, even in the absence of formal security instruction. By characterizing the skills and strategies students already employ, educators can better design interventions that cultivate a security mindset through familiar programming tasks.

To this end, we investigate the following **research question**: *What skills and strategies do undergraduate students without formal security training use when attempting to identify vulnerabilities in software?*

We conducted a qualitative study with 15 undergraduate computer science students using two think-aloud sessions in which participants first secured a text-based game and then attempted to attack peers' implementations. For most students, this was their

first formal exposure to computer security, so the tasks were designed to make threats concrete and approachable. Students drew on existing skills and prior experiences, including programming practices and informal security exposure. Although all participants identified at least one vulnerability, fewer carried out successful attacks, and many stopped after a single discovery, illustrating how students engage with adversarial problems when security is embedded in familiar programming contexts.

Our analysis suggests that even brief exposure to defend–attack cycles can prompt threat perception and adversarial reasoning using existing knowledge; however, students often struggle to move beyond initial vulnerability identification. We characterize recurring patterns in threat conceptualization, the reuse of non-security strategies (e.g., testing and edge-case reasoning), and decisions to stop exploring the attack space. We conclude with instructional implications and propose lightweight, scalable approaches to foster a security mindset within core computer science courses.

2 Related Work & Background

BIBIFI & CTFs: Capture the Flag (CTF) and Build-It, Break-It, Fix-It (BIBIFI) are competition formats used to teach and engage students in security. CTFs are typically attack-oriented: participants attempt to obtain a string of text known as a 'flag.' CTF challenges often require cryptography knowledge, network security knowledge, and sometimes "simultaneous defending," which can overwhelm novices [40].

BIBIFI challenges have three distinct phases in which participants *build* software from provided specifications, *break* others' implementations by finding bugs, and finally *fix* any bugs discovered in their software [28]. BIBIFI activities always include defending; their format separates these elements and mirrors parts of real-world security life cycles and red-team/blue-team practice. BIBIFI has also been adopted in a handful of security classes (e.g., [25]).

Undergraduate Security Education: Security education research in post-secondary settings predominantly has two focus areas: (1) computer security-specific contexts, such as security courses or cybersecurity programs; and (2) integration across the general CS curriculum. The latter aims to teach concepts early and often without causing major disruption to existing coursework [6, 32]; our work is situated in this second area.

Computer security education often covers *secure coding practices*, such as using real-world software vulnerabilities to improve self-efficacy and awareness [10]. Alternative approaches apply these practices in non-security courses [4, 14], incorporate assistive tools into IDEs [38], create drill-and-practice platforms [5], and use adaptive tutoring systems [22].

To help alleviate instructors' workload, researchers have created security-based labs on upper-level topics (e.g., network security) [11] and topics suitable for CS0/CS1/CS2 courses [33]. Others have developed and implemented assignments based on security scenarios [21, 27]. These assignments often focus on a specific security concept and offer detailed guidance and instruction on the intended actions. Others have looked towards and analyzed traditional learning materials and found instances of insecure code and lack of discussion of security in database textbooks [34] and computer systems courses' lecture materials [3].

This body of literature largely examines outcomes by evaluating students' "correctness" or recall of facts using simple pre-/post-assessments. These assessments often target well-known and prevalent vulnerabilities, including those that frequently contribute to real-world attacks and data leaks [19]. However, this piecemeal approach does not necessarily help students build more generalizable, *adaptable* security thinking. Much of the existing curriculum also tacitly assumes that learning is a one-way transmission process. The learning sciences, however, show that students are *active* participants in learning, with their own knowledge, experiences, and preconceptions shaping their *construction* of knowledge [9]; this perspective is known as constructivism. CS and programming education research often leverages constructionism, a sub-theory of constructivism; one well-known example is Papert's LOGO [24].

Recent work has taken a constructivist approach to undergraduate security education, including a concept inventory used to detect misconceptions in students who have taken at least one formal security course [26]. We previously found that undergraduate CS students without prior formal security coursework can demonstrate instances of a security mindset with minimal prompting [17]. A UK study investigated CS undergraduate and graduate students' perceptions, experiences, and practices with S&P [31]. Building on this work, we examine how students with limited formal security experience approach attacks on peers' versions of a system similar to one they previously built.

3 Methods

Our study investigates how students' existing thinking can inform the design of tools that foster a security mindset. By building on what students *already* understand, interventions may be integrated into crowded CS curricula without major restructuring. We do not focus on the correctness or efficiency of solutions, nor do we expect students to explore all possibilities. Instead, we examine how they apply their current skills to a security-focused challenge.

We adapted the "Build-It / Break-It" framework from BIBIFI competitions [28]. Tasks were designed for students who had completed a CS2 course[1] and involved securing and attacking a text-based adventure game. The game tracks player progress—level, health, items, upgrades—using the Java class `GameProgress`, which maintains game state and handles passcode generation and processing. Passcodes encode the game state, similar to a save file (Fig. 1).

3.1 Study Protocol & Data Collection

Each participant completed two one-hour think-aloud sessions over Zoom, separated by 7–10 days (with a few longer gaps). These sessions captured *in situ* security reasoning. Participants could use any external resources or tools except GenAI. In the first session (Build-It), we collected participants' code both as data and for use in the second session. We fixed compilation issues in advance and then supplied an assistive tool (Sec. 3.3) in the second session (Break-It); logs and reports were collected. Across both sessions, we recorded screen video and audio, which were transcribed with supporting notes and screenshots. Sessions concluded with a semi-structured interview. After completing both sessions, participants filled out a survey capturing demographics, coursework, and prior security

[1] Covering object-oriented programming and basic data structures.

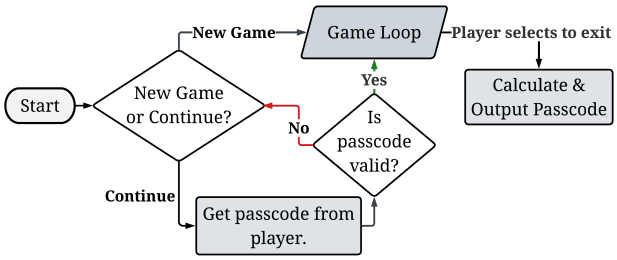

Figure 1: A state diagram of the game's passcode system, showing two potential attack surfaces: the passcode entry and the game loop. Students were not explicitly informed about these surfaces.

experience; collecting this at the end helped reduce stereotype-related bias during the sessions. All procedures were approved by our institution's IRB.

3.2 Build-It, Break-It Tasks

The first session was a "Build-It" phase. Participants were assigned the role of a game developer for a simple text-based adventure game and tasked with designing their own passcode system and defending the game from cheating players. We also explicitly announced an upcoming attack phase to prompt security thinking, which students may not engage in spontaneously [31], while intentionally leaving the task open-ended to assess their threat perception. Participants were provided with a folder of working starter code for the game, an explanation of passcode systems, and a list of the development team's established game design elements. The prompt also suggested that they focus on the `GameProgress` class, which is responsible for processing passcodes and maintaining the game state. This was included to prevent participants from spending the entire hour finding where to begin. However, this did not prevent them from looking elsewhere in the code or playing the game in its current state.

The second session of the study was the "Break-It" phase, in which participants assumed the role of a cheating player—the opposite of their role in the previous session—and identified vulnerabilities. Different avenues were available, including playing the game, using a passcode generator, or examining provided source code. Participants were given a folder with anonymized source code for implementations to break and an executable JAR for an assistive tool, "VDRT," described in Section 3.3. We selected a total of five `GamePasscodes` from Build-It sessions for the Break-It phase. Each fell into one of three categories denoted by ♣, ♡, and ♠:

(1) **PLAIN-RE ♣:** Rule-enforced plaintext passcode (enforces game rules, e.g., $currentHealth \leq maxHealth$)
(2) **PLAIN-RT ♣:** Plaintext passcode but limits integer input values for health restore event
(3) **SHIFT ♠:** Simple Caesar cipher (shifts each character by a fixed amount)
(4) **WMAP ♡:** Substitution mapping (one-to-one mapping of values to words, e.g., Lucky Charm → sweat); includes clamping for valid value ranges
(5) **MATH-ENC ♡:** Mathematical encoding, where students applied arithmetic transformations to encode the game state (e.g., $levelNum \times 7 : upgrades \times 13 : currentHealth^4$). The encoding appends a level-item suffix, "wkcwd" if the player possesses the required item for the level, and "r73kdm" otherwise. Checks that entered passcodes contain valid values.

These implementations were representative of the majority of approaches observed in the Build-It phase. Two Break-It sets, A and B, were created by selecting one implementation from each of the three categories (♣, ♡, ♠), with precautions taken to avoid assigning participants their own implementation. Set A included PLAIN-RE, WMAP, and SHIFT and was assigned to eight participants (P1–2, 4–5, 10–12, 14), while Set B included MATH-ENC, SHIFT, and PLAIN-RT and was assigned to seven participants (P3, 6–9, 13, 15).

3.3 Vulnerability Discovery and Reporting Tool

To facilitate the Break-It session, we developed a vulnerability discovery & reporting tool (VDRT). Because this may have been the first time participants attacked a program, the tool was designed to minimize technical overhead so they could focus on attack reasoning. To work seamlessly with the Build-It prompt, the tool was built in Java and used the Swing library for the GUI. The tool provided four main functions: (1) switching between implementations, (2) playing the game via a dedicated window, (3) a passcode generation oracle (based on the selected implementation), and (4) a form for reporting vulnerabilities.

Implementation Switching: Using reflection and class loaders, VDRT can load and run different bytecode. This allows users to switch between multiple implementations of the same class(es) to explore and exploit. The distribution includes a file that denotes which implementation files belong to each set. The first window at launch has buttons labeled with each set name. Pressing a button moves to the other windows with a dropdown menu, where the selected implementation is the one loaded and run in (2) and (3).

Passcode Oracle: The tool includes a *passcode oracle*, which allows users to specify game-state values and generate (or attempt to generate) a passcode. A table of levels and required items is provided for reference (see Fig. 2). The oracle was included to save time, preventing students from wasting time trying to reach specific game states for passcodes. It also reduces the monotony of this process,[2] while still providing an authentic, interactive experience [7, 12]. While a real attacker would not have such a "magical box," passcode-sharing is common in games via online forums or in-person, and students naturally considered this type of knowledge during threat modeling in the Build-It phase.

Each time a passcode is generated, the oracle logs the passcode, set name, implementation, game-state values, and timestamp. If a passcode cannot be generated, the resulting error message is recorded instead.

Game Window: The game itself provided another potential attack surface. The game window (Fig. 3) allows participants to play using a selected implementation from a dropdown menu. Students could explore the game to discover vulnerabilities through normal gameplay or to verify the effectiveness of passcode-related attacks.

[2]Random elements in the game can make it difficult to reach the same state consistently.

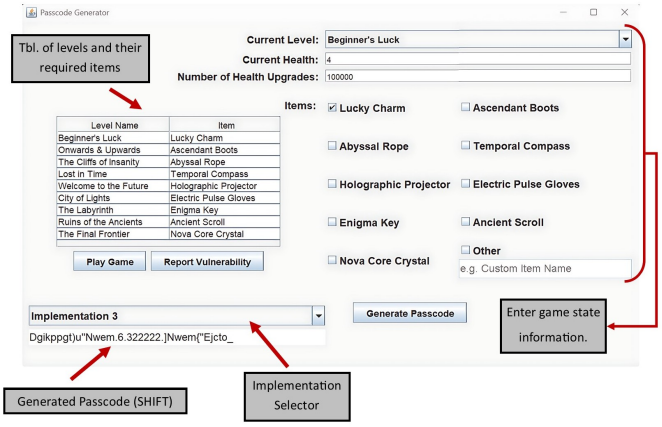

Figure 2: A screenshot of the passcode oracle that displays a generated passcode for SHIFT. Note that the passcode is for an invalid game state due to the number of health upgrades.

Vulnerability Reporting: In ethical hacking, a key output of vulnerability discovery is reporting details to the appropriate entity. VDRT includes a form to report vulnerabilities and provides insight into *what* participants do and do not consider vulnerabilities. Since students may not have formal experience with bug reporting, this form provides a structured way for them to describe findings. We analyzed bug bounty programs (BBPs) from major tech companies (i.e., Google, Meta, Microsoft) and a sample from HackerOne [1], a popular BBP host. Our form recorded three common report fields: the implementation (product), actions taken to reproduce the issue (reproducibility steps), and the outcome (what it does). Other common fields were omitted because the required security knowledge was beyond the expected student level. Fields for expected outcome, discovery strategy, and additional comments were included to gather further data about participants' thinking.

3.4 Participant Information & Recruitment

We recruited 15 students from an R1 university in the United States through flyers and a department mailing list. Interested students completed a screening survey asking for their declared/intended major and completion of a CS2 course or equivalent. Those who indicated CS as their major and had completed a CS2 course were invited to participate. We initially encountered challenges in matching gender distribution to the site's population. To address this, we added a gender question to the screening survey to support selective sampling. Seven additional participants did not complete Session 2; their Session 1 code may be used, but their data are excluded from analysis.

The participants included 11 male students, 1 female student, 1 non-binary student, and 1 person who opted not to disclose their gender.[3] With regard to program year, there were 4 first-year students, 7 second-year students, 1 third-year student, and 3 fourth-year students. Additional majors included data science (6), information science (1), mathematics (1), statistics (1), computer engineering (1), legal studies (1), and psychology (1). One of them (P2) noted a formal security course (cryptography), and others noted relevant experiences such as self-study, reading articles/YouTube videos, guided hacking tutorial sites with gamification elements (e.g., TryHackMe), breaking games as a kid, cybersecurity club meetings, brief in-class mentions, and informal programs in high school.

[3]The female and non-binary students were recruited after the screening survey adjustment.

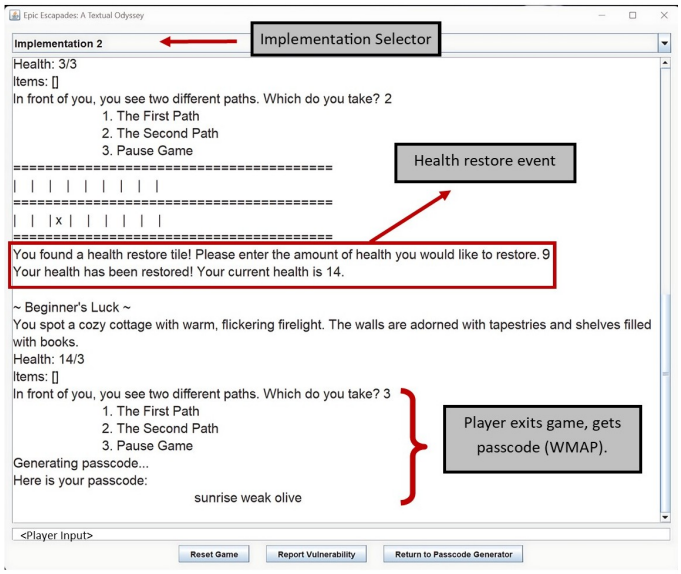

Figure 3: A screenshot of the game being played through the tool. The game is using the WMAP implementation and has allowed the player to restore above their max health.

3.5 Analysis

We processed the passcode logs with a program that generated a change log, which was then merged with the corresponding Break-It transcript by timestamp for easy reference. To inform future educational tools, we conducted an inductive qualitative analysis informed by Sherin's constructivist assessment work [30]. We performed an initial pass through Session 2 (Break-It) data, coding patterns in successive passcode generations, atypical inputs to the game or passcode generator, and atypical gameplay. The think-aloud protocol also surfaced "thought processes" (e.g., reasoning for actions, stated strategies, and information sources), which we coded as well. After the first pass, the research team discussed initial codes; themes were then aggregated into parent codes. This process was repeated iteratively until no new codes emerged. The final code set was then reapplied to the full data set by one researcher.

4 Results

We organize the findings around recurring attack strategies and knowledge sources observed in participants' actions and think-aloud comments.

4.1 Attacks Discovered

Participants identified multiple attacks in the game implementations, with varying levels of success. We observed two main attack categories: (1) passcode manipulation, which involves crafting invalid passcodes to trigger unintended behavior; and (2) gameplay

manipulation, which involves exploiting unsanitized input during health point (HP) restore events to gain infinite HP.

All participants focused primarily on cracking the passcode system. Once they understood an encoding scheme, participants could often outline potential attacks that a cheating player might perform, such as gaining extra health. Twelve participants (all except P5, P9, and P10) tested hypothesized attacks with varying success, intentionally entering invalid passcodes generated by the oracle or manipulated manually. For example, P6 successfully entered "63:25:256:r73kdm" for the MATH-ENC implementation, which should be rejected because 25 is not a multiple of 13. As P12 explained:

> *"...if you know how to make a bad [cheating] passcode but the game [is] not gonna let you put in that bad passcode, then I guess it's not really a vulnerability... in the end you're not going to get the gain you want." – P12 (1^{st} year, male)*

Number of Attacks Considered. Some participants focused narrowly on a single vulnerability. For example, P5 explored only the HP restore event. More than half of the participants (P1–3, 7–10, 13) considered a single attack category and typically moved on to a different implementation once they identified one vulnerability, even though additional categories were also testable in principle. The remaining participants (P4–6, 11, 12, 14, 15) investigated multiple attack surfaces; all of them exploited the HP restore event, and P14 additionally targeted a game mechanic to gain infinite health upgrades. In summary, all 15 participants attempted to find vulnerabilities in the passcode system, while only 7 also attempted to find them in the gameplay loop.

4.2 Systematic Testing

All participants used systematic, hypothesis-driven testing as part of their approach. A common strategy was to modify one passcode parameter at a time (e.g., health points, level, upgrades, items) while holding the others constant. Participants described this process in several ways, including formal scientific methods such as controlled (P9) or independent-variable (P12) experiments, decomposition strategies such as divide-and-conquer (P9), pattern recognition (P3–5, 13), general problem-solving (P1, 5), experimentation (P5), and unit-test-style approaches (P10, 14). One participant summarized this process as follows:

> *"I would say it was like writing some kind of unit test, black-box unit test... it was coming up with some kind of bad input to figure out what's wrong with the implementation. Yeah, I think that's the same process." – P10 (4th year, male)*

A central aspect of systematic testing was probing invalid inputs and edge cases. Using the passcode oracle, participants experimented with invalid health point (HP) values (P2–5, 7, 9–10, 12, 14), out-of-range upgrades (P1–5, 7, 9–10, 12), items from future levels (P2–3, 5–6, 8), and custom items (P3, 5–6, 8, 11, 12, 14). Participants applied similar strategies in the gameplay loop, particularly by exploiting unsanitized input during HP restore events (P4–6, 11–12, 14–15). P14 explicitly targeted edge cases to trigger integer overflow, which could permit invincibility by reducing health below zero ($HP = 0$). Notably, only P1 and P8 consulted the provided source code to confirm hypotheses.

Taken together, these behaviors resemble *reverse engineering*, even if participants did not use the term explicitly. Reverse engineering is a common technique for identifying vulnerabilities [15, 39]. In this context, students reconstructed aspects of system behavior despite lacking formal training in advanced static analysis or specialized tools such as Ghidra.

4.3 Leveraging Prior Experiences

Participants frequently transferred prior knowledge into their attack strategies. We observed three recurring sources of prior experience: their own earlier Build-It work, formal coursework, and activities outside class.

From Prior Build-It Work: Nine participants explicitly stated that their earlier defense implementation shaped how they approached attacks. Six (P2–3, 7–8, 13–14) referenced concrete system rules or design details, while others (P1–3, 5, 7, 11, 13–14) described a broader transfer of defensive thinking into offensive testing. As P14 described:

> *"I mean, part of it was just how last time, when I was doing the build-it part, I noticed that health [values] could be a vulnerability while trying to program it. ...just having that in mind already, I feel like it was just intuitive to me." – P14 (2^{nd} year, male)*

This pattern suggests that reflecting on defenses may have helped participants anticipate likely weaknesses and prioritize where to probe first.

From Formal Coursework: Participants also drew on classroom knowledge. P2 referenced cracking practices from a cryptography course. Several participants cited general programming knowledge (P1, 8–9, 14) as useful, and P8 (1^{st} year, male) used knowledge about C strings terminating with null bytes when attempting an overflow saying, "something weird happens with null bytes and strings, maybe that would lead to something". As noted earlier, P1 and P4 used knowledge of numeric representation in memory to target integer-overflow conditions. Three participants (P4, 10, 14) also applied software-engineering practices, including unit testing (P10, 14), debugging (P4), and edge-case analysis (P4).

From Informal and Personal Contexts: Experiences outside formal coursework also informed strategy selection. The most common source was prior engagement with games: directly cheating (P1–2, 8, 10–11) or observing exploits in others' gameplay (P11, 14). The extent of transfer appeared to vary by game type: some experiences, such as Pokèclicker (P8) and CandyBox2 (P1), appeared directly applicable, whereas others, such as MMORPG play (P14), appeared to support broader problem-solving habits. Participants also referenced puzzles and mystery-oriented activities, including childhood "mystery phases" (P13), ciphers in media (P3), escape rooms (P7), and ARGs (P11). Three participants (P6–7, 11) cited online resources such as security tutorials or YouTube recommendations. Additional examples included personal server administration (P6), experimenting with tools such as CyberChef,[4] and bypassing restrictions in freemium software (P11).

Overall, 13 participants referenced prior knowledge, spanning all program years and a wide range of completed coursework. We did not observe clear differences by year or coursework background: the

[4] https://cyberchef.org/

two participants who did not explicitly reference prior experience (P12, 15) did not share an obvious demographic or curricular profile. These observations suggest that transfer in this task may not map cleanly to seniority or course progress alone.

5 Discussion

Our results show that students rely heavily on skills and strategies they already possess, such as systematic testing, debugging, and problem decomposition, to reason about vulnerabilities. These existing skills provide a foundation for introducing security concepts without requiring major changes to an already crowded CS curriculum. Educators can build on these capabilities by framing testing and edge-case analysis not only as debugging techniques but also as entry points for exploring security vulnerabilities. For example, asking students to design tests that anticipate malicious input or abnormal game states can reinforce the connection between program correctness and security reasoning.

Prior Experiences as Cognitive Scaffolds: Both formal and informal prior experiences influenced participants' approaches. Students leveraged prior defending exercises, classroom coursework, and activities outside the classroom—such as playing or cheating in games, or solving puzzles—to guide their attack strategies. Defending prior implementations appeared to prime students for vulnerability reasoning, even without explicit instruction. Similarly, programming knowledge, unit-testing skills, and understanding of memory were frequently applied in the attack tasks. This aligns with prior work showing that professional hackers and testers can use similar processes to discover vulnerabilities [37]. Outside experiences provided transferable problem-solving heuristics that enriched students' exploration, though their applicability likely depends on task context. Together, these findings suggest that diverse experiences can serve as cognitive scaffolds for developing a security mindset [16].

Designing Instructional Scaffolds: Our findings suggest that incorporating defend–attack cycles into coursework could help foster a security mindset. Even minimal prior exposure to defending code, such as through the Build-It phase, encouraged students to anticipate potential attacks [17], indicating that these activities can act as effective scaffolds. To maximize learning, instructors could integrate short defend–attack tasks in existing programming assignments, framing familiar practices such as testing and edge-case analysis not only as debugging techniques but also as avenues for identifying security vulnerabilities. The open-ended nature of attack tasks further supported students in exploring multiple solution paths, enabling them to practice flexible and adaptable reasoning. Together, these interventions leverage students' existing skills while encouraging adversarial thinking, helping bridge the gap between standard programming practice and security mindset development.

Tool Support for Classroom Integration: VDRT provides a flexible, nascent framework for implementing defend-attack tasks in classroom settings. By including features such as (passcode) oracles, reporting assistance, and source code viewing, the tool could be used in the future to scaffold students' engagement and provide immediate feedback on attacks. However, the current version is a prototype and future development should have configurable prompts and could allow selective enabling of features as students develop skills, gradually increasing challenge and complexity to support progressive learning.

Comprehensiveness in Vulnerability Discovery: We observed tension between focused and broad exploration. Approximately half of the participants considered only a single attack category per implementation, often moving to a different implementation once a vulnerability was identified. Others explored multiple categories, sometimes uncovering subtler vulnerabilities. Prior work has shown that even students with security experience will fail to consider a range of vulnerabilities [35]. The pattern we observed may reflect the task framing, which did not explicitly ask students to find *all* vulnerabilities or provide incentives to report more than one. It may also indicate that breadth of exploration depends on factors such as intrinsic curiosity or prior defensive focus. In practice, however, defensive work requires attention to multiple vulnerability categories. Future interventions could guide students to consider multiple attack surfaces by prompting reflection on overlooked categories or offering incentives for systematic exploration, while maintaining equity for underrepresented students.

Limitations: The notable limitations are the single-institution context and potential self-selection bias. The design may have increased participants' focus on passcode-related vulnerabilities, although this focus also appeared in most Build-It sessions, which partially mitigates that concern. The time limit is another limitation, though it also functioned as a tradeoff by forcing participants to prioritize solutions. Lastly, although all authors contributed to the qualitative codebook, the final coding pass was not subjected to external inter-rater reliability measures.

6 Conclusions

We conducted a think-aloud study in which undergraduate CS students defended and attacked a video game with minimal security prompting to examine how students without formal security training approach adversarial tasks. Our goal was to understand students' existing skills to inform the design of curricula, tools, and assignments that integrate security thinking into core CS courses. Our findings show that students relied on systematic testing strategies and prior experiences from coursework, informal learning, and gameplay. While most participants identified at least one vulnerability and attempted an attack, many stopped after a single issue despite opportunities to continue exploring. This suggests that minimal defend–attack exposure may elicit adversarial reasoning, but additional scaffolding is likely needed to support broader exploration of the attack space. Based on these results, we identify skills that can be leveraged in instructional design and outline directions for expanding VDRT and developing lightweight interventions that support sustained security mindset development in undergraduate CS education.

Acknowledgments

We thank the reviewers of our paper for their insightful comments and suggestions. This research was supported in part by NSF award #2350165.

References

[1] [n. d.]. HackerOne. https://www.hackerone.com/.

[2] Amrita Ahuja. 2024. https://www.staffingindustry.com/news/global-daily-news/us-needs-nearly-265000-additional-cybersecurity-workers

[3] 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)*. 271–281. doi:10.1145/3372782.3406266

[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)*. 973–979. doi:10.1145/3545945.3569730

[5] Leo St. Amour and Eli Tilevich. 2025. Designing a Platform to Train Secure Programming Skills with Attack-and-Defend Exercises. In *2025 IEEE Global Engineering Education Conference (EDUCON)*. 1–10. doi:10.1109/EDUCON62633.2025.11016492

[6] 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)*. 82–88. doi:10.1145/3341525.3387426

[7] Michael Buckley, Helene Kershner, Kris Schindler, Carl Alphonce, and Jennifer Braswell. 2004. Benefits of using socially-relevant projects in computer science and engineering education. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*. 482–486.

[8] 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

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

[10] Denise Daniels, Joon Suk Lee, Hui Chen, and Kostadin Damevski. 2024. Utilizing Real-World Software Vulnerabilities to Enhance Secure Programming Education. In *2024 IEEE Frontiers in Education Conference (FIE)*. 1–8. doi:10.1109/FIE61694.2024.10893584

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

[12] Katrina Falkner and Edward Palmer. 2009. Developing authentic problem solving skills in introductory computing classes. In *Proceedings of the 40th ACM technical symposium on Computer science education*. 4–8.

[13] Tiago Espinha Gasiba, Samra Hodzic, Ulrike Lechner, and Maria Pinto-Albuquerque. 2021. Raising Security Awareness using Cybersecurity Challenges in Embedded Programming Courses. *arXiv* (2021). https://arxiv.org/abs/2102.10436 arXiv:2102.10436.

[14] 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. doi:10.1109/ICCQ51190.2021.9392965

[15] Biswaroop Guha and Biswanath Mukherjee. 1997. Network security via reverse engineering of TCP code: Vulnerability analysis and proposed solutions. *IEEE Network* 11, 4 (1997), 40–48.

[16] Gonca Irmak and Konstantinos Kaldaras. 2024. Employing technology-enhanced feedback and scaffolding to support the development of deep science understanding using computer simulations. *International Journal of STEM Education* 11 (2024), 30. doi:10.1186/s40594-024-00490-7

[17] 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 *Proceedings of the 2025 ACM Conference on International Computing Education Research V.1 (ICER '25)*. 16–26. doi:10.1145/3702652.3744226

[18] Amruth N. Kumar, Rajendra K. Raj, Sherif G. Aly, Monica D. Anderson, Brett A. Becker, Richard L. Blumenthal, Eric Eaton, Susan L. Epstein, Michael Goldweber, Pankaj Jalote, Douglas Lea, Michael Oudshoorn, Marcelo Pias, Susan Reiser, Christian Servin, Rahul Simha, Titus Winters, and Qiao Xiang. 2024. *Computer Science Curricula 2023*.

[19] MITRE. 2025. 2025 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2025/2025_kev_list.html

[20] Azqa Nadeem. 2023. Cybersecurity as a Crosscutting Concept Across an Undergrad Computer Science Curriculum: An Experience Report. *arXiv* (2023). https://arxiv.org/abs/2310.07625 Also published at ACM/educational venues, DOI:10.1145/3626252.3630821.

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

[22] Ida Ngambeki, Matt Bishop, Jun Dai, Phillip Nico, Shiven Mian, Ong Thao, Tran Ngoc Bao Huynh, Zed Chance, Isslam Alhasan, and Motunrola Afolabi. 2022. SecTutor: An Intelligent Tutoring System for Secure Programming. In *Information Security Education - Adapting to the Fourth Industrial Revolution*, Lynette Drevin, Natalia Miloslavskaya, Wai Sze Leung, and Suné von Solms (Eds.). 17–28.

[23] 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)*. 966–972. doi:10.1145/3545945.3569782

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

[25] Bryan Parno. 2020. Project 1: Build it; break it; fix it. https://course.ece.cmu.edu/~ece732/s20/homework/bibifi/SPEC.html

[26] 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. doi:10.1145/3451346

[27] Vahab Pournaghshband and Hassan Pournaghshband. 2024. Entailing Security Mindset in Foundational CS Courses: An Interactive Approach. In *2024 IEEE World Engineering Education Conference (EDUNINE)*. 1–6. doi:10.1109/EDUNINE60625.2024.10500581

[28] 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. https://www.usenix.org/conference/cset15/workshop-program/presentation/ruef

[29] Lwin Khin Shar, Christopher M. Poskitt, Kyong Jin Shim, and Li Ying Leonard Wong. 2022. XSS for the Masses: Integrating Security in a Web Programming Course using a Security Scanner. In *arXiv*. https://arxiv.org/abs/2204.12416 arXiv:2204.12416.

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

[31] Mohammad Tahaei, Adam Jenkins, Kami Vaniea, and Maria Wolters. 2021. "I Don't Know Too Much About It": On the Security Mindsets of Computer Science Students. In *Socio-Technical Aspects in Security and Trust*, Thomas Groß and Theo Tryfonas (Eds.). 27–46.

[32] 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)*. 320–324. doi:10.1145/1352135.1352246

[33] 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). doi:10.1145/2897441

[34] 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)*. 191–197. doi:10.1145/3287324.3287429

[35] 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.

[36] Alina Torbunova, Adnan Ashraf, and Ivan Porres. 2024. A Systematic Mapping Study on Teaching of Security Concepts in Programming Courses. *arXiv* (2024). https://arxiv.org/abs/2407.07511 arXiv:2407.07511.

[37] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. 2018. Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes. In *2018 IEEE Symposium on Security and Privacy (SP)*. 374–391. doi:10.1109/SP.2018.00003

[38] Michael Whitney, Heather Richter Lipford, Bill Chu, and Tyler Thomas. 2018. Embedding Secure Coding Instruction Into the IDE: Complementing Early and Intermediate CS Courses With ESIDE. *Journal of Educational Computing Research* 56, 3 (2018), 415–438. arXiv:https://doi.org/10.1177/0735633117708816 doi:10.1177/0735633117708816

[39] Reginald Wong. 2018. *Mastering Reverse Engineering: Re-engineer your ethical hacking skills*. Packt Publishing Ltd.

[40] Valdemar Švábenský, Pavel Čeleda, Jan Vykopal, and Silvia Brišáková. 2021. Cybersecurity knowledge and skills taught in capture the flag challenges. *Computers & Security* 102 (2021), 102154. doi:10.1016/j.cose.2020.102154