International Journal of Computer Science Education in Schools, December 2021, Vol. 5, No. 2 ISSN 2513-8359

Good examples help; bad tools hurt: Lessons for teaching computer security skills to undergraduates

Jonathan Sharman, Claudia Ziegler Acemyan, Philip Kortum, Dan Wallach

Rice University, USA

DOI: 10.21585/ijcses.v5i2.131

Abstract

Software security is inevitably dependent on developers' ability to to design and implement software without security bugs. Perhaps unsurprisingly, developers often fail to do this. Our goal is to understand this from a usability perspective, identifying how we might best train developers and equip them with the right software tools. To this end, we conducted two comparatively large-scale usability studies with undergraduate CS students to assess factors that affect success rates in securing web applications against cross-site request forgery (CSRF) attacks. First, we examined the impact of providing students with example code and/or a testing tool. Next, we examined the impact of working in pairs. We found that access to relevant secure code samples gave significant benefit to security outcomes. However, access to the tool alone had no significant effect on security outcomes, and surprisingly, the same held true for the tool and example code combined. These results confirm the importance of quality example code and demonstrate the potential danger of using security tools in the classroom that have not been validated for usability. No individual differences predicted one's ability to complete the task. We also found that working in pairs had a significant positive effect on security outcomes. These results provide useful directions for teaching computer security programming skills to undergraduate students.

Keywords: Security, coding, teaching, usability, tools

1. Introduction

Despite a growing emphasis among security experts on secure coding practices, software developers continue to regularly misuse or misunderstand secure coding tools. Understanding how to best train students in good security coding practices is critical to designing safer software. Recent efforts within the area of usable security research have attempted to enumerate causes for developer error leading to security vulnerabilities in software. For example, access to good documentation and reliable example code have significant impacts on solving security tasks (Acar et al., 2017; Fischer et al., 2017; Mindermann and Wagner, 2018; Mindermann and Wager, 2020), as does priming (Naiakshina et al., 2018). Usability issues can also impact the appropriate use of other security-related systems, including Android development (Acar et al., 2016), cryptographic APIs (Acar et al., 2017; Acar et al., 2017; Gorski et al., 2018; Naiakshina et al., 2019; Oliveira et al., 2018; Zeier et al., 2019), type systems (Weber et al., 2017), HTTPS deployment (Krombholz et al., 2017; Bernhard et al., 2019), OpenSSL (Ukrop and Matyas, 2018), and string and I/O APIs (Oliveira et al., 2018).

Our work builds on previous studies by trying to understand how to better instruct undergraduate computer science students in the art of security programming, examining the impact of using code samples, software tools, and group programming in the classroom. We wanted to determine if there were benefits in providing students with software code examples and software tools that would aid them in testing for code security. We also wanted to determine if working in teams as students was beneficial in their learning efforts. We chose cross-site request forgery (CSRF) as a security problem for our study because it's still relevant to current practice and because of its relative simplicity for use as a teaching tool. We performed two between-subjects usability studies, one year apart, with undergraduate computer science students. In study 1 we examined the impacts of example code and a CSRF detection tool on a student's ability to repair CSRF vulnerabilities in a test server. In study 2, we examined the effects of the students working alone vs. working in pairs.

2. Study 1: Impact of example code and software tools

2.1 Methodology

We drew our students from the sophomore-level COMP 215 class at Rice University, a highly-selective four-year residential college. Rice University had a total sophomore population of approximately 1,000 students, and Computer Science is the most popular major on campus, with approximately 19% of 2018 sophomores enrolling in COMP 215. COMP 215 students present a relatively uniform pool of students. Most COMP 215 students have never taken a college-level course in Java before, having come to Rice University immediately after high school. (Only eight students among the students in this study were transfer students.) All COMP 215 students take the same freshman class sequence, including an introductory course in Python, and a theory course. To help prepare students for the security task, the COMP 215 lectures during the week of the experiment considered "Web 2.0" designs (e.g., Java microservices with JavaScript clients). Students were strongly encouraged to attend a security-specific lecture as well.

2.1.1 Design

This study used a between-subjects design with two variables: (1) availability of a CSRF testing tool and (2) availability of an example web application (code) with CSRF mitigation, resulting in four experimental conditions. The "no tool, no example" group, which served as the control group, had access to neither CSRF Testing Tools nor to the example code. The "example-only" group had access to the example code but not to the tool. The "tool-only" group had access to the tool but not to the example code. Finally, the "tool and example" group had access to both the tool and the example code. Students were randomly assigned into one of the four conditions. Students in all groups, including those not provided with example code, were allowed to search the Internet for examples and instructions but not to ask anyone for help.

We selected CSRF prevention as the task for this study because it is a realistic security problem but also relatively simple, allowing us to teach the relevant concepts to our students over a few lectures and construct a short, self-contained study based on the problem. In contrast, many other security vulnerabilities, such as buffer overflows, can be very subtle to understand or even recognize when looking at the code.

Condition Assignment. Consistent with the other programming assignments in COMP 215, students attempted this task individually. (We examine pair programming in Study 2, below.) We controlled for gender and midterm grades when assigning students to conditions, to reduce possible confounding factors and to allow us to check for any effects these variables had on learning outcomes. We partitioned the remaining students by gender and sorted both lists by midterm grade. We then pulled students in blocks of four and distributed them at random among the four groups. Students who did not identify as male or female were assigned to groups at random. Despite a small number of drop-outs after experimental group assignment, this strategy yielded a consistent demographic distribution, with 44 participants assigned to the control group, 46 to the "example-only" group, 45 to the "tool-only" group, and 45 to the "tool and example" group. Students were instructed not to discuss or share any aspect of the assignment with each other, including their condition assignment, and since the project was graded only on participation, they had no academic incentive to disclose their condition assignment to other students. Average age of the students was 19.2 years (SD 0.8). Each of the groups was composed of approximately 28% females. Java experience was consistent across groups, with an average of 2.1 on a 5-point scale, where 5 is expert (SD 1.0).

The dependent variables in this study were effectiveness (how well the students completed the task) and efficiency (how long it took them to complete the task) in implementing CSRF prevention in a test server. For groups with access to the CSRF Testing Tool, we also measured satisfaction with the provided tool, i.e. how well the tool met the student's expectations. Effectiveness, efficiency, and satisfaction are three standard measures used to assess system usability per ISO 9241-11.

2.1.2 Measures

Effectiveness. Effectiveness is the degree of success in achieving a goal (ISO 9241-11). Consistent with previous assignments in the course, we assigned each student's work a security score from 0 to 10 using a subtractive grading rubric (Table 1). Deductions were capped at 10 points; i.e., a solution with 10 or more deductions received a security score of 0. Most mistakes resulted in a one-point deduction, with the exception of failing to employ any kind of server-provided key, which resulted in a four-point deduction.

One of the most common anti-CSRF techniques consists of sending a randomized token from the server to the client, returning that token from the client to the server with each request, and verifying on the server that the tokens match. Critically, the client must not return the token via a cookie since cookies associated with a domain are sent automatically with each request to that domain. While students were permitted to use whatever method they wanted to complete the task, this is the technique students were taught during lecture and the approach they generally took.

Table 1. Security score rubric

Error	Deduction
Server vulnerable to GET-based CSRF	1
No server-generated key	4
Key entropy < 32 bits	1
Key entropy < 64 bits	1
No handshake mechanism to prevent back doors	1
Vulnerable to timing attacks	1
Server does not attempt to send key	1
Client does not attempt to retrieve key	1
Server-to-client methods do not match	1
Key not delivered server-to-client	1
Client does not send key for validation	1
Server does not request key for validation	1
Client-to-server methods do not match	1
Key not sent client-to-server	1
Client sends key insecurely (e.g. via a cookie)	1
Server does not validate received key	1

We initially considered a scoring system based only on the presence of actual vulnerabilities; however, we decided that such a system did not capture how well each student understood the problem or how secure their solution was. For example, many students submitted solutions that were correct except that they did not use a one-time key to prevent a fraudulent web form from retrieving the CSRF-prevention key, leaving the server vulnerable. These students might receive a zero score from a classical security analysis, despite demonstrating significant progress towards a secure solution. Therefore, we instead chose a scoring system based on the accumulation of individual errors. While students were free to implement any CSRF mitigation mechanism they chose, the vast majority used techniques similar to those described in both the lecture materials and the example code.

Additionally, we assigned each student a passing or failing functionality score based on whether they broke the original functionality of the system. For example, some students made it impossible to perform any transactions at all. For the sake of simplicity, we graded solutions with minor changes in functionality (such as reduced responsiveness) as functionally correct.

Timing. We set the maximum duration for the study to 180 minutes (measured to the nearest minute), opting for this duration both to respect the students' time and to enable most students to complete the study in a single sitting. The time on task does not include reading the project specification prior to beginning the task or the post-study survey. We obtained timing data in two ways, in an attempt to improve the reliability of the timing data: self-reported time on task and git push/commit times. Even so, it was difficult to obtain high-quality timing data under our experimental setup. In cases where the two measurements did not align or where the available timing data was clearly wrong (for instance, a full day spent on the task, based on push times), we had to discard the data for the purpose of timing analysis.

Surveys. We used SurveyMonkey for both the demographic and post-study surveys. For those in the experimental groups that had access to CSRF Testing Tools, the post-study survey includes a System Usability Scale (SUS) (Brooke, 1996) evaluation in the final survey to measure satisfaction with the tool. The SUS is one of the most commonly used, psychometrically validated

(Bangor et al., 2008) tools that measure the usability of products, services, and systems. SUS scores range from zero (unusable) to 100 (highly usable).

2.1.3 Materials

Spark Java Server. Spark Java[1] is a simple Java web framework. We wrote a Spark Java server and client for a toy application called "Fruit Market", which simulates buying and selling of fruit with fluctuating prices. Students run both the server and client locally, changing the server state by clicking the buy and sell buttons. We intentionally left the server open to CSRF attacks by implementing the buying and selling functionality using simple GET requests. For instance, a purchase of a certain type of fruit can be made by issuing a GET request to `/buy/?index=0`, and a sale of the same fruit can be made through `/sell/?index=0`. Since there is no CSRF mitigation mechanism, the application is vulnerable to all the standard CSRF attacks, such as embedding these URLs in an HTML image element. Note that Spark Java does not include any CSRF prevention mechanism.

Handouts / GitHub Classroom. GitHub Classroom allows instructors to create assignments from template git repositories, which students clone when they click on a given link. We created four project PDFs and four Classroom clone links, corresponding to our four experimental conditions, written in the same style as our regular weekly project assignments.

Lectures. We presented a three-lecture sequence considering how web browsers and servers operate, along with the security issues they face. We provide PDFs of all lecture slides on our course web page.

Example Code. Half of the students had access to an example web application, already secure against CSRF attacks. The example server, also written in Spark Java, implemented the back-end for a JavaScript read-eval-print-loop (REPL). The JS REPL server avoids CSRF by launching the user's browser with a one-time key in the URL, which the client then returns to the server in order to retrieve a CSRF-prevention key, generated randomly at server launch. This key then serves to validate subsequent calls to the server.

CSRF Testing Tools. Half of the students had access to CSRF Testing Tools[2], a free, semi-automated CSRF exploit generation tool, along with the instructions provided by the tool's author. The tool consists of two parts: a "FormGrabber" bookmarklet that can be used to copy HTML form content from a webpage into the user's clipboard and a "FormBuilder" page that creates a spoofed form from the copied form content. While the README describes how to use the tool to "create [HTML] forms that mimic the forms on the site that you're testing", it does not explain how to interpret the results and whether they indicate a CSRF vulnerability. Indeed the README states "I am assuming that you have a good understanding of what a CSRF attack is and can figure out how this tool mimics one. Explaining the anatomy of a CSRF attack is not something I'm going to do in this documentation." In short, the ability to change server-side application state via interaction with the spoofed form indicates a CSRF vulnerability.

Students had no prior experience with this tool in the course but were instructed to read the included README describing the steps required to set up and utilize the tool to check for CSRF vulnerabilities. As part of its setup process, CSRF Testing Tools requires (1) hosting a JavaScript file on a server separate from the server under test and (2) modifying a bookmarklet file to point to the hosted file. We gave the students a copy of the latest version of the tool as found on GitHub, with one modification: we hosted the file for them and gave them a pre-modified version of the bookmarklet. While performing this modification ahead of time changes the experience of using the tool, we determined that requiring the students to complete this step themselves would be unreasonably difficult and time-consuming within the limits of the study. As our results show, the tool's measured effectiveness and satisfaction were low, even with this simplification to the process.

We considered several other free CSRF prevention tools—including Burp Suite[3], CSRFTester[4], Pinata[5], CSRF PoC Generator[6], and OWASP Zed Attack Proxy (ZAP)[7]—but in our judgment, CSRF Testing Tools was the most useful among these options.

An anonymized repository with experiment materials can be found at https://github.com/bad-tools-hurt/csrf. This repository contains the code, handouts, and lectures used in both studies, with all references to the authors and their institution(s) removed. We have removed code that is both irrelevant to the task and unique to the course. In particular, for this repository, we replaced a course-specific JSON library with a third-party library.

[1] https://sparkjava.com/
[2] https://github.com/akrikos/CSRF-Testing-Tools
[3] https://portswigger.net/burp
[4] https://github.com/tomasperezv/web-security-tools/tree/master/CSRFTester
[5] https://github.com/ahsansmir/pinata-csrf-tool
[6] https://security.love/CSRF-PoC-Genorator/
[7] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

2.1.4 Procedures

As with any human subjects experiment, we obtained IRB approval before beginning the study. We recruited a total of 194 students from the fall 2018 COMP 215 class. Students received partial course credit for participating (regardless of their success) and had the ability to opt out of the study at any point prior to or during the study. None of the students opted out; however, a total of 14 students either dropped the course prior to the completion of the study or otherwise did not complete the study, leaving a total of 180 students in the data set.

Once the assignment was given, the students could choose to attempt the three-hour task any time within a five-day period. We encouraged the students to attempt the task in one sitting, but they were allowed to take breaks as long as they noted the stop and start times via git commit messages. The students had to complete the following steps: (1) read the project specification PDF; (2) click the GitHub Classroom assignment link (embedded in the PDF), to set up their repository; (3) add their name and student ID to a README file prior to beginning the task and then commit and push to GitHub (the timestamp of this initial push allowed us to measure the start of the time on task); (4) secure the test server against CSRF attacks, by whatever method they choose, within 180 minutes of their initial push time; (5) after completing the task, running into the time limit, or giving up on the task, commit and push their final code; and (6) fill out the post-study survey.

We chose to have the students work on their own time in a setting of their choosing, rather than in a lab, for better scalability and to allow the students to complete the task under their usual work conditions, making comparisons with past course performance more meaningful. The first author manually graded each submission according to our security and functionality rubrics. We used the CSRF Testing Tools to speed up the grading process; however, it was only useful as a first step towards measuring security. For instance, many students came very close to a correct solution but did not use a handshake mechanism, such as a one-time key, to prevent a fraudulent web form from reading the anti-CSRF key from the HTML. Under our rubric, such a solution is graded as 9/10, but its security-relevant behavior is identical to a solution with no CSRF mitigation at all. Conversely, some solutions were invulnerable to the particular attack generated by CSRF Testing Tools but used insufficient key entropy and thus still lost points. Therefore, we still had to manually inspect each submission in order to understand whether a solution was fully secure and, if not, how many points should be deducted.

2.2 Results and Discussion

We performed an ANOVA for security score, functionality score, and time on task, with experimental condition assignment (group) and gender as factors, using Tukey's HSD to correct for multiple comparisons.

Security Scores. The mean security score was 1.43 for the control group, 4.15 for "example-only", 1.53 for "tool-only", and 2.71 for "tool and example" (see Figure 1). There was a reliable effect of group assignment on security scores ($F(3,173) = 6.05$, $MSE = 69.42$, $p < .01$, $\eta^2 = .09$), but there was insufficient evidence of a reliable effect of gender on security scores ($F(1,173) = .55$, $MSE = 6.35$, $p = .46$, $\eta^2 < .01$). Tukey's HSD (Table 2) indicates significant differences only for "example-only" to the control (diff. of means = 2.68, $p < .01$) and "example-only" to "tool-only" (diff. of means = 2.58, $p < .01$). Cohen's d for mean "example-only" security score vs. control group security score is .77, which indicates a medium effect size. Cohen's d for "example-only" vs. "tool-only" is .73, also a medium effect size.

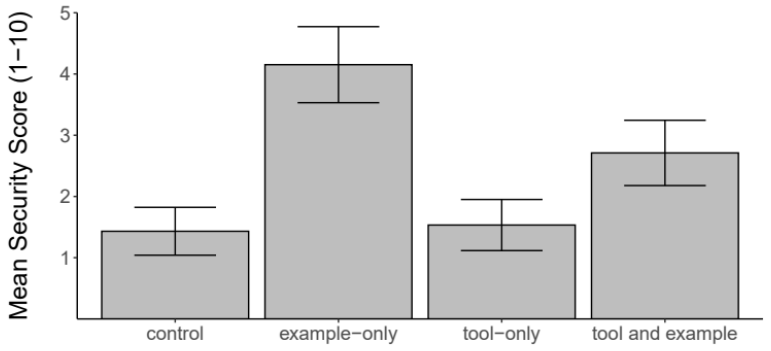

Figure 1. Mean security scores by study 1 group, with error bars depicting the standard error of the mean.

Table 2. Comparison of mean security scores by group, study 1 (Tukey HSD).

Comparison	Difference of Means	p

Example-only - Control	2.68	<.01
Tool-only - Control	.10	>.99
Tool and Example - Control	1.28	.29
Example-only - Tool-only	2.58	<.01
Example-only - Tool and Example	1.40	.21
Tool and Example - Tool-only	1.18	.35

Security scores were highly variable across all experimental conditions. A large number of students failed to make any significant progress towards a secure solution (103 out of the 180 students received zero security points). Overall, security scores were quite low, and very few students completely secured the application against CSRF (0% in the control group, 24% in "example-only", 0% in "tool-only", and 13% in "tool and example").

Despite common perceptions, prior work suggests that CS grades are typically unimodal (Patitsas et al., 2016). Consistent with past research, Hartigan's dip test fails to reject the hypothesis that COMP 215 grades are unimodal ($D = .02$, $p = .93$). However, there is significant evidence of non-unimodality in the "example-only" security scores ($D = .12$, $p < .01$).

Functionality Scores. The mean functionality scores were .59 for the control group, .63 for "example-only", .67 for "tool-only", and .49 for "tool and example" (see Table 3). There was a reliable effect of gender on functionality ($F(1,173) = 6.89$, $MSE = .55$, $p = .01$, $\eta^2 = .04$), but there was insufficient evidence of a reliable effect of group on functionality ($F(3,173) = 1.31$, $MSE = .30$, $p = .27$, $\eta^2 = .02$). Table 4 shows the differences of mean functionality scores between the groups, none of which showed evidence of a reliable effect. For male vs. female functionality scores, difference of means = .21, $p = .01$, and Cohen's d is .43, indicating a small effect.

Table 3. Functionality scores (0 or 1), study 1.

Group	*n*	Mean	*SD*
Control	44	.59	.50
Example-only	46	.63	.49
Tool-only	45	.67	.48
Tool and Example	45	.49	.51

Table 4. Comparison of mean functionality scores by group, study 1 (Tukey HSD).

Comparison	Difference of Means	*p*
Example-only - Control	.07	.91
Tool-only - Control	.08	.88
Tool and Example - Control	.10	.75
Example-only - Tool-only	.01	>.99
Example-only - Tool and Example	.17	.34
Tool and Example - Tool-only	.18	.30

Time on Task. Timing data is unavailable for a total of eleven students (two from the control group and three each from the other groups). There were three types of student error that led to missing or excluded timing data: failure to commit and push code before starting the task, large unexplained discrepancies between reported time and time measured via git pushes, and breaks taken without reporting the duration. The mean time on task in minutes (Table 5) was 155.7 for the control group ($n = 42$, $SD = 32.83$), 149.1 for "example-only" ($n = 43$, $SD = 34.05$), 151.3 for "tool-only" ($n = 42$, $SD = 41.21$), and 148.1 for "tool and example" ($n = 42$, $SD = 38.59$). We did not observe evidence of a reliable effect.

Table 5. Time on task, study 1.

Group	*n*	Mean	*SD*
Control	42	155.7	32.83
Example-only	43	149.1	34.05
Tool-only	42	151.3	41.21
Tool and Example	42	148.1	38.59

Multiple Linear Regression. To gain a better understanding of factors that may contribute to success in repairing CSRF vulnerabilities, we performed multiple linear regression on the security scores of each experimental condition, with time on task, final grade in COMP 215, GPA, number of the three security lectures attended, and self-reported Java experience (rated from 1-5) as possible factors (Table 4). In the control group, adjusted $R^2 = .24$, $F(5,30) = 3.24$, $DF = 30$, and $p = .02$. Final grade was the only significant factor. In "example-only", adjusted $R^2 = .32$, $F(5,33) = 4.55$, $DF = 33$, and $p < .01$, and the only significant factor was time on task. In "tool-only", adjusted $R^2 = .28$, $F(5,32) = 3.80$, $DF = 32$, and $p < .01$, and the significant factors were lectures attended and Java experience. Finally, in "tool and example", adjusted $R^2 = .22$, $F(5,32) = 3.03$, $DF = 32$, and $p = .02$, and there were no significant factors.

Table 6. Multiple linear regression, security scores ("Tool" = tool-only, "Ex." = example code only, "Both" = tool and example code).

FACTOR	η^2				***t***				***p***			
	Control	Ex.	Tool	Both	Control	Ex.	Tool	Both	Control	Ex.	Tool	Both
Time on Task	.05	.14	<.01	.05	1.41	.56	.09	1.49	.17	.02	.93	.15
Final Grade	.15	.07	.03	.07	2.48	1.80	1.07	1.68	.02	.08	.29	.10
GPA	<.01	<.01	.01	.02	.07	.03	.75	.88	.95	.98	.46	.39
Lectures Attended	.05	.02	.11	.03	1.43	.88	2.28	1.03	.16	.38	.03	.31
JAVA Experience	.05	.04	.15	.05	1.47	1.37	2.59	1.45	.15	.17	.01	.16

2.2.1 Post-task Survey Results

CSRF Testing Tools. SUS Scores In the "tool-only" group, the mean SUS score for CSRF Testing Tools was 42.94 ($n = 40$, $SD = 19.78$). In the "tool and example" group, it was 39.10 ($n = 39$, $SD = 19.97$). We did not observe a reliable effect of example code availability on SUS scores, based on Welch's *t*-test ($DF = 77$, $t = .86$, $p = .39$). According to the adjective rating scale developed by Bangor et al. (Bangor et al., 2009), these scores fall between "poor" (mean score of 35.7) and "OK" (mean score of 50.9), indicating a very low degree of usability compared to other systems in general (though not necessarily to other CSRF mitigation tools). Moreover, it is rare in practice to find SUS scores below 40 for complex multi-step tasks, such as CSRF mitigation (Kortum and Acemyan, 2013).

Security and Functionality Confidence. We asked participants to rate their confidence in their solution in terms of security and functionality, each on a scale from 1 (low) to 5 (high). Note that self-reported task completion and confidence did not affect participants' participation credit in the course. The mean security confidence was 2.00 in the control group ($n = 37$, $SD = 1.11$), 2.82 in "example-only" ($n = 38$, $SD = 1.27$), 1.93 in "tool-only" ($n = 40$, $SD = 1.27$), and 2.03 in "tool and example" ($n = 39$, $SD = 1.20$). The mean functionality confidence was 2.65 in the control group ($n = 37$, $SD = 1.57$), 2.92 in "example-only" ($n = 38$, $SD = 1.65$), 2.38 in "tool-only" ($n = 40$, $SD = 1.53$), and 2.23 in "tool and example" ($n = 39$, $SD = 1.49$).

To measure how well students were able to gauge their own performance, we examined the correlations between confidence and actual scores, for both security and functionality. For security, the correlation was .59 in the control group ($t = 4.31$, $DF = 35$, $p < .01$), .60 in "example-only" ($t = 4.48$, $DF = 36$, $p < .01$), .50 in "tool-only" ($t = 3.57$, $DF = 38$, $p < .01$), and .68 in "tool and example" ($t = 5.66$, $DF = 37$, $p < .01$). For functionality, the correlation was .38 in the control group ($t = 2.45$, $DF = 35$, $p = .02$), .50 in "example-only" ($t = 3.45$, $DF = 36$, $p = .01$), .21 in "tool-only" ($t = 1.31$, $DF = 38$, $p = .20$), and .65 in "tool and example" ($t = 5.15$, $DF = 37$, $p < .01$). In summary, there was a significant and moderately large correlation between confidence and actual scores in all cases except for functionality in the "tool-only" group.
With regard to perceived security, arguably the most dangerous situation is one in which a developer believes strongly that they

have protected the system against CSRF when they have not. In this situation, not only does the system remain vulnerable, but the developer is unlikely to seek help to repair the vulnerability. To quantify this condition, we define the "false confidence" rate as the proportion of students whose submission had a security score of zero for which (1) the student reported that they had completed the task and (2) the student rated their confidence in the security of the system as either a four or five on a 1-5 scale. The false confidence rates were .035 for the control group (n = 28), 0 for "example-only" (n = 16), .107 for "tool-only" (n = 28), and .053 for "tool and example" (n = 19). The false confidence rates are fairly low across the board, which is a reassuring result. We did not observe a reliable effect of experimental condition on the rate of false confidence.

3. Study 2: Impact of team programming

Study 2 examined the impact of working in pairs on the learning process. We recognized that the most successful condition ("example-only") from study 1 gave the students example code, but no CSRF tooling, so we used that as the starting point in study 2.

3.1 Methodology

We drew our students from the 2019 COMP 215 class at Rice University. The 2019 curriculum for this course was similar to that of 2018, with the notable exception of the introduction of partners for weekly projects, while the 2018 class projects were all performed solo. In 2019, beginning in week 7, students partnered with a different classmate of their choosing for each weekly project. The CSRF project occurred during week 11, after three such partnered projects, the same point in the semester as in study 1.

3.1.1 Design

Our goal for study 2 was to understand the impact of single versus two-person teams, so we used a between-subjects design with one variable: "Solo" vs. "Duo". We provided both groups access to example code and neither group access to CSRF Testing Tools (i.e., the "example-only" condition). The dependent variables remain the same as in study 1, with the exception of SUS results for CSRF Testing Tools since it was unused.

Condition Assignment. Students were randomly assigned into one of the two conditions, subject to two constraints. First, consistent with COMP 215 team policy, no two students were assigned to work on a team who had previously worked together. Several COMP 215 students had special exemptions from this rule and were allowed to work with the same partner each week; these students completed the assignment but are excluded from the study results. Second, we asked any students who may have difficulty collaborating during the study (e.g., due to travel) to inform us so that they could be assigned to work alone. Twelve students out of a total of 163 informed us of such difficulties and were assigned to the Solo group. The remaining 151 students were split into the two groups uniformly at random. In total, the Solo group had 53 students, and the Duo group had 110. Note that there are approximately twice as many individuals in the Duo condition as in the Solo condition, in order to maintain an approximately equal numbers of teams.

For logistical reasons, it was not practical to keep it a secret from the students that some of them were working alone and some with a partner. Therefore, unlike in study 1, participants in study 2 were aware of their condition assignment. In the Solo group, student ages ranged from 18 to 21 years (Mdn = 19, $\bar{X}$= 19.3, SD = .696). Sixteen students identified as female and 36 as male, and one gave no response. Self-reported Java experience prior to taking COMP 215 ranged from 1 to 4 (Mdn = 2, $\bar{X}$= 2.132, SD = .981). In the Duo group, ages ranged from 18 to 26 years, ($\bar{X}$ = 19.4, Mdn = 19, SD = 1.160). There were 28 female students, 81 male, and one identifying as neither male nor female. Self-reported Java experience ranged from 1 to 5 (Mdn = 2, $\bar{X}$= 1.926, SD = .924).

Timing. Unlike study 1, we did not examine git push times. This would have been infeasible for the Duo condition, where each student has a separate time-on-task and either student may push to the repository at any time. Instead, we simply asked each student to report their personal time on-task in minutes, working with or without their partner, excluding any breaks. We used the same timing reporting scheme for the Solo condition, for consistency.

3.2 Results and Discussion

Security Scores. The mean security score was 2.57 for Solo and 4.95 for Duo (Figure 2), and there was evidence of a reliable effect (t= 2.94, DF= 105, p< .01). Cohen's d= .56, indicating a medium effect size. As in study 1, security scores were variable but generally low, with 52 of 108 students receiving zero security points. Only 25% of Duo teams and 17% of solo teams fully secured the application against CSRF. Hartigan's dip test indicates non-unimodality in both Solo (D= .087, p < .01) and Duo (D= .161, p < .01) security scores. As in 2018, the 2019 COMP 215 final grades exhibit no reliable non-unimodality (D= .025,

p = .70). There was a reliable effect of group assignment ($F(1,101) = 9.47$, MSE=161.07, $p < .01$) but not of team gender composition ($F(3,101) = 1.91$, MSE=32.56, $p = .13$) on security scores. We also note that despite the "example-only" group from study 1 and the Solo group from study 2 having similar experimental conditions, "example-only" had a mean security score of 4.15 vs. 2.56 for the Solo group. However, despite the apparently worse performance of the Solo group, there was insufficient evidence of a reliable effect ($t = 1.90$, DF = 94, $p = .06$), and the effect size is small (Cohen's d = .38).

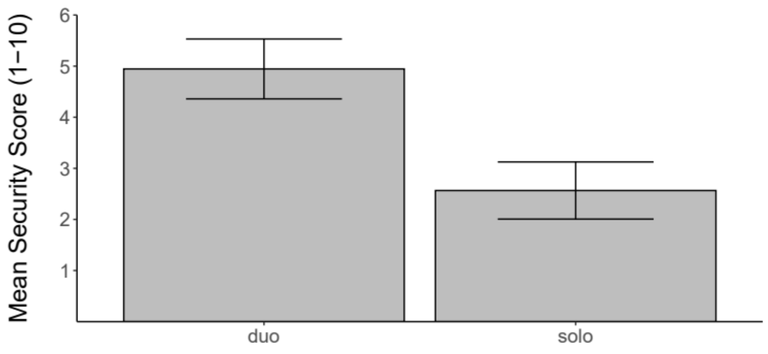

Figure 2. Mean security scores by study 2 group, with error bars depicting the standard error of the mean.

Functionality Scores. The mean functionality scores (Table 7) were .62 for Solo and .67 for Duo. There was insufficient evidence of a reliable effect on functionality scores of either team gender composition ($F(3,101) = 1.80$, MSE = .41, p = .15) or group ($F(1,101) = .57$, MSE = .13, p = .45).

Table 7. Functionality scores (0 or 1), study 2.

Group	*n*	Mean	*SD*
Solo	53	.62	.49
Duo	55	.67	.47

Time on Task. Timing data is unavailable for a total of six teams (four from Solo and two from Duo), due to students neglecting to complete the post-task survey. When only one student in a Duo team reported the time on task, we take that figure as the team average. The mean time on task in minutes (Table 8) was 181 for Solo (n = 49, SD = 37.6) and 169 for Duo (n = 53, SD = 39.8).

Table 8. Time on task, study 2.

Group	*n*	Mean	*SD*
Solo	49	181	37.6
Duo	53	169	39.8

Multiple Linear Regression. We performed multiple linear regression once again to try to determine which factors we examined contribute most to success in repairing CSRF vulnerabilities. As before, we used time on task, final grade in COMP 215, GPA, number of the three security lectures attended, and self-reported Java experience (rated from 1-5) as factors (Table 9). For the Duo condition, since there are two students per submission, we used the team average for each factor. In Solo, adjusted R^2 = .32, $F(5,40) = 5.24$, DF = 40, and $p < .01$. The significant factors were time on task, final grade, and lectures attended. In Duo, adjusted R^2 = .061, $F(5,47) = 1.67$, DF = 47, and p = .16, and no factors showed evidence of a reliable effect.

Table 9. Multiple linear regression, Solo and Duo security scores.

FACTOR	η^2		t		p	
	Solo	Duo	Solo	Duo	Solo	Duo
Time on Task	.09	.04	2.40	1.47	.02	.15
Final Grade	.17	.04	3.30	1.53	<.01	.13
GPA	.02	<.01	1.22	.44	.23	.67

Lectures Attended	.09	.01	2.34	.69	.02	.49
JAVA Experience	.01	.02	.63	1.13	.53	.26

3.2.1 Post-task Survey Results

Security and Functionality Confidence. We again asked students to rate their security and functionality confidence from 1-5. The mean security confidence was 2.25 (SD 1.15) in Solo and 2.91 (SD1.18) in Duo. The mean functionality confidence was 2.71 (SD 1.53) in Solo versus 3.19 (SD 1.41) in Duo.

Tables 10 and 11 show the security and functionality confidence correlations for study 2. For security, the correlation was .60 in Solo and .33 in Duo. For functionality, the correlation was .29 in Solo and .57 in Duo. There was a significant and moderately large positive correlation between confidence and actual scores in all cases in study 2. Using the same definition of "false confidence" as in study 1, the rates in study 2 were 0 for Solo (n = 35) and .147 for Duo (n = 34). Five individuals from three different Duo groups exhibited false confidence, whereas no Solo students did. This time, experimental condition did have a reliable effect on the false confidence rate ($F(1,61) = 5.86$, MSE = .41, p = .02, η^2 =.09). Cohen's d is .58, a medium effect size. Interestingly, in exactly one team with a security score of zero, there was a mismatch of confidence levels, with one partner highly confident (4) and one not (zero).

Table 10. Confidence-security correlation, study 2.

Group	Pearson's *r*	*t*	*DF*	*p*
Solo	.60	5.20	47	<.01
Duo	.33	3.46	97	<.01

Table 11. Confidence-functionality correlation, study 2.

Group	Pearson's *r*	*t*	*DF*	*p*
Solo	.29	2.10	47	.04
Duo	.57	6.82	97	<.01

We were surprised to see that the Duo group had a higher false confidence rate. One potential explanation is that participants working in teams tended to rely on and trust their partners, assuming the teammate knew more than they actually did. This result may be cause for some concern and caution when working collaboratively, but in terms of security results, the Duo teams still performed much better overall.

4. General Discussion

Our results identify important factors for improving student success rates in learning how to repair CSRF vulnerabilities, as well as some factors that surprisingly had little or no effect. These results also include a baseline SUS score for any future usability studies on CSRF prevention tools, which is valuable for drawing standard and objective comparisons between the satisfaction such tools provide users.

Consistent with our expectations and with previous research (see, e.g., Acar et al., 2017), access to reliable example code had a significant impact on the security of students' solutions, at least when not coupled with CSRF Testing Tools. Students in the "example-only" group had moderately higher security scores than both the control group and the group with access to the tool alone. These results confirm the efficacy of quality example code in training students to improve code security. Also in line with expectations, student teams of two produced significantly more secure solutions than students working alone. To the best of our knowledge, this study is the first to empirically demonstrate that working in pairs can result in better outcomes when training for computer security tasks.

Contrary to our expectations, access to CSRF Testing Tools did not reliably improve security scores. Even more surprisingly, *the tool actually appears to have hurt the security scores for students with example code*. We can only postulate what went wrong. One possibility is that the students were constrained by the time limit, and the tool distracted students from spending time understanding and adapting the example code. We know that in at least some cases, the tool actively misled students into thinking an insecure solution was secure. One student commented in a git commit message that they were done because "[the page] runs on FormBuilder"; i.e., the spoofed form generated by the tool worked, and the student interpreted that as an indication that the server was secure even though it actually implies the exact opposite. This is an easy mistake to make for a developer lacking computer security expertise when using a tool with little documentation. Furthermore, among the 79 students with access to CSRF Testing Tools who completed the post-task survey, nine (11.4%) specifically mentioned understanding the tool as one of the most difficult parts of the assignment.

In absolute terms, the scores we obtained in study 1 indicate that CSRF Testing Tools is not perceived to be usable by the students. However, we cannot conclude that CSRF Testing Tools is necessarily worse in this regard than other CSRF prevention tools, since this is the first study to apply the SUS to this class of tools. However, researchers and tool authors can compare the satisfaction of existing and future products against this baseline freeware tool, providing empirical evidence for their comparative usability. Anecdotally, while a few students commented that the tool was helpful in solving the task, many other students indicated that they did not understand how to begin using the tool or how to interpret its behavior. We conjecture that more detailed documentation, including a description of what to expect when the CSRF vulnerability has or has not been patched, could improve perceived usability.

Students sometimes erroneously think their code is secure when it is not. In the field, this false sense of security can result in security defects making their way into deployed software, putting end users at risk. Previous studies have found differing results regarding the correlation between how secure developers think their code is and how secure it actually is (Acar et al., 2017; Gorski et al., 2018). In this study, we found that students' perceptions of their code's functional correctness and security generally matched their actual performance. It is possible that these correlations were driven by the appreciable number of students who failed to make any significant progress towards a solution and in that case knew for certain that they had failed. We were surprised to see that the Duo group had a higher false confidence rate. One potential explanation is that participants working in teams tended to rely on and trust their partners, assuming the teammate knew more than they actually did. This result may be cause for some concern and caution when working collaboratively, but in terms of security results, the Duo teams still performed much better overall.

We expected that students with higher GPAs and/or final grades in the course would produce more correct and more secure solutions. However, GPA did not have a significant effect, and final grades in COMP 215 were only predictive in the study 1 control and study 2 Solo groups. We also expected prior experience with Java to have a positive impact on security outcomes, but we only observed a significant effect of self-reported Java experience on security secures in the "tool-only" group.

4.2 Limitations

We limited our students' time on task to 180 minutes. The primary motivation for restricting the time on task was to prevent students from wasting too much time on the assignment if they became stuck. A secondary motivation was to allow us to encourage the students to complete the task in a single contiguous block of time, reducing errors in recorded times (e.g., if students failed to report breaks). Unfortunately, since many students used the entire allotted time, the time restriction makes it difficult to distinguish between students who truly became stuck and students who could have made additional significant progress if given more time. One possible mitigation would be to have a prior week's assignment using the exact same codebase, only without security considerations, and thus give the students additional familiarity with the experimental setup. We could also try to further simplify the security task, although this particular task is already quite simple, at least for students who understand the problem.

Lecture attendance had a *negative* correlation with security outcomes in every group except "example-only", and the effect was significant in the "tool-only" and Solo groups. This effect is inconsistent, and one possible explanation is that stronger students might not bother attending lectures, in general. However, it is also entirely possible that our security lectures did not do a good job of explaining CSRF, and that many students solved the problem simply by transferring the coding pattern from the example without understanding how it works or why it is important. Prior research has shown that copying and pasting code can lead to security bugs (Fischer et al., 2017), underscoring the importance of reliable code samples but also of security awareness and critical thinking on the part of developers. A future study might provide more insight into students' security comprehension, in addition to security performance, by including a series of security questions before and after the assignment, to gauge students' security skills "on paper".

5. Conclusions and Future Work

In this work, we conducted two large studies with undergraduate computer science students to investigate which factors affect students' ability to successfully complete a computer security assignment, viz. securing a web app against cross-site request forgery. In study 1, we examined the impact of providing students with particular resources during the assignment: a fuzz-testing tool and/or example code. In study 2, we looked at the effects of working alone vs. in a pair.

We found that providing students with both example code and a software tool does not appear to confer the additive benefit to learning outcomes we expected. Our results establish the benefit of access to example code when attempting to teach CSRF prevention but also demonstrate that an unusable tool can not only fail to significantly improve learning how to implement good security outcomes but actually be detrimental to success in some cases. The results of our follow-up study further demonstrate that working with a partner has substantial benefits to learning outcomes when compared with working alone. Prior work both in education and industry has demonstrated that collaborative programming increases confidence and satisfaction, reduces frustration, improves course and major retention, and otherwise improves outcomes (Williams and Upchurch, 2001; Williams et al., 2002; Mcdowell et al., 2002; Mcdowell et al., 2003; Werner et al., 2004; McDowell et al., 2006; Eierman and Iversen, 2018; Williams et al., 2000; Williams et al., 2002; Hanks et al., 2004; Smith et al., 2018; Werner et al., 2004; McDowell et al., 2006; Arisholm et al., 2007; Begel and Nagappan, 2008; Hannay et al., 2009). As such, it's unsurprising but important to note that working in pairs also improves educational outcomes in training students in security measures.

Based on these results, we suggest three key recommendations for instructors with regard to teaching students how to implement secure code: (1) seek out trustworthy example code to give to students for applying security coding techniques, (2) be wary of tools in the classroom whose efficacy is unverified, and (3) when possible, allow students to collaborate in solving these kinds of security challenges, rather than working alone. Although we used CSRF prevention as our example, it's quite likely that these findings apply to teaching many other classes of security challenges being taught in the classroom. In sum, example code helps. Partners help. Bad tools hurt.

References

Acar, Y., Backes, M., Fahl, S., Garfinkel, S., Kim, D., Mazurek, M. L., and Stransky, C. (2017). Comparing the Usability of Cryptographic APIs. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 154-171. https://doi.org/10.1109/SP.2017.52

Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M. L., and Stransky, C. (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)*, pages 289-305. https://doi.org/10.1109/SP.2016.25

Acar, Y., Stransky, C., Wermke, D., Mazurek, M. L., and Fahl, S. (2017). Security Developer Studies with GitHub Users: Exploring a Convenience Sample. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 81-95, Santa Clara, CA. USENIX Association. https://www.usenix.org/system/files/conference/soups2017/soups2017-acar.pdf

Arisholm, E., Gallis, H., Dyba, T., and Sjoberg, D. I. K. (2007). Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering, 33*(2):65-86. https://doi.org/10.1109/TSE.2007.17

Bangor, A., Kortum, P., and Miller, J. (2009). Determining what individual sus scores mean: Adding an adjective rating scale. *Journal of Usability Studies, 4*(3):114-123. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.177.1240&rep=rep1&type=pdf

Bangor, A., Kortum, P. T., and Miller, J. T. (2008). An empirical evaluation of the system usability scale. *International Journal of Human-Computer Interaction, 24*(6):574-594. https://doi.org/10.1080/10447310802205776

Begel, A. and Nagappan, N. (2008). Pair programming: What's in it for me? *In Proceedings of the Second ACMIEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '08*, page 120-128, New York, NY, USA. Association for Computing Machinery. https://doi.org/10.1145/1414004.1414026

Bernhard, M., Sharman, J., Acemyan, C. Z., Kortum, P., Wallach, D. S., and Halderman, J. A. (2019). On the Usability of HTTPS Deployment. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI '19*, pages 310:1-310:10, New York, NY, USA. ACM. https://doi.org/10.1145/3290605.3300540

Brooke, J. (1996). SUS: A quick and dirty usability scale. Usability evaluation in industry, page 189.

Eierman, M. and Iversen, J. (2018). Comparing testdriven development and pair programming to improve the learning of programming languages. *Journal of the Midwest Association for Information Systems*, 2018:23-36. https://doi.org/10.17705/3jmwa.000038

Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., and Fahl, S. (2017). Stack Overflow Considered Harmful? The Impact of Copy Paste on Android Application Security. In 2017 *IEEE Symposium on Security and Privacy*, pages 121-136. https://doi.org/10.1109/SP.2017.31

Gorski, P. L., Iacono, L. L., Wermke, D., Stransky, C., Moeller, S., Acar, Y., and Fahl, S. (2018). Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse. In *SOUPS @ USENIX Security Symposium.* https://www.usenix.org/system/files/conference/soups2018/soups2018-gorski.pdf

Hanks, B., Mcdowell, C., Draper, D., and Krnjajic, M. (2004). Program quality with pair programming in cs1. *In ACM Sigcse Bulletin,(36)*, pages 176-180. https://doi.org/10.1145/1026487.1008043

Hannay, J., Dybå, T., Arisholm, E., and Sjøberg, D. (2009). The effectiveness of pair programming: A meta-analysis. *Information and Software Technology, 51*:1110-1122. https://doi.org/10.1016/j.infsof.2009.02.001

ISO 9241-11 (2018). *Ergonomics of human-system interaction - Part 11: Usability: Definitions and concepts.*

Kortum, P. and Acemyan, C. Z. (2013). How low can you go?: Is the system usability scale range restricted? *Journal of Usability Studies, 9*(1):14-24. http://tecfa.unige.ch/tecfa/maltt/ergo/1415/UtopiaPeriode4/articles/JUS_Kortum_November_2013.pdf

Krombholz, K., Mayer, W., Schmiedecker, M., and Weippl, E. (2017). 'I Have No Idea What I'm Doing' - On the Usability of Deploying HTTPS. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1339-1356, Vancouver, BC. USENIX Association. https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-krombholz.pdf

Mcdowell, C., Werner, L., Bullock, H., and Fernald, J. (2002). The effects of pair-programming on performance in an introductory programming course. In *ACM SIGCSE Bulletin,(34)*, pages 38-42. https://doi.org/10.1145/563340.563353

Mcdowell, C., Werner, L., Bullock, H. E., and Fernald, J. (2003). The impact of pair programming on student performance, perception and persistence. In *25th International Conference on Software Engineering*, 2003, pages 602-607. https://doi.org/10.1109/ICSE.2003.1201243

McDowell, C., Werner, L., Bullock, H. E., and Fernald, J. (2006). Pair programming improves student retention, confidence, and program quality. *Communications of the ACM, 49*(8):90-95. https://doi.org/10.1145/1145287.1145293

Mindermann, K. and Wagner, S. (2018). Usability and Security Effects of Code Examples on Crypto APIs - CryptoExamples: A platform for free, minimal, complete and secure crypto examples. *2018 16th Annual Conference on Privacy, Security and Trust (PST)*. https://doi.org/10.1109/PST.2018.8514203

Mindermann, K. and Wagner, S. (2020). Fluid Intelligence Doesn't Matter! Effects of Code Examples on the Usability of Crypto APIs. *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 306-307. https://doi.org/10.1145/3377812.3390892

Naiakshina, A., Danilova, A., Gerlitz, E., and Smith, M. (2020). On conducting security developer studies with cs students: Examining a password-storage study with cs students, freelancers, and company developers. *In Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1-13. https://doi.org/10.1145/3313831.3376791

Naiakshina, A., Danilova, A., Gerlitz, E., von Zezschwitz, E., and Smith, M. (2019). 'If You Want, I Can Store the Encrypted Password': A Password-Storage Field Study with Freelance Developers. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI '19*, pages 140:1-140:12, New York, NY, USA. ACM. https://doi.org/10.1145/3290605.3300370

Naiakshina, A., Danilova, A., Tiefenau, C., and Smith, M. (2018). Deception task design in developer password studies: Exploring a student sample. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 297-313, Baltimore, MD. USENIX Association. https://www.usenix.org/system/files/conference/soups2018/soups2018-naiakshina.pdf

Oliveira, D. S., Lin, T., Rahman, M. S., Akefirad, R., Ellis, D., Perez, E., Bobhate, R., DeLong, L. A., Cappos, J., and Brun, Y. (2018). API Blindspots: Why Experienced Developers Write Vulnerable Code. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 315- 328, Baltimore, MD. USENIX Association. https://www.usenix.org/system/files/conference/soups2018/soups2018-oliveira.pdf

Patitsas, E., Berlin, J., Craig, M., and Easterbrook, S. (2016). Evidence that computer science grades are not bimodal. In ICER '16: Proceedings of the *2016 ACM Conference on International Computing Education Research*, pages 113-121. https://doi.org/10.1145/2960310.2960312

Smith, M. O., Giugliano, A., and DeOrio, A. (2018). Long term effects of pair programming. *IEEE Transactions on Education, 61*(3):187-194. https://doi.org/10.1109/TE.2017.2773024

Ukrop, M. and Matyas, V. (2018). Why Johnny the Developer Can't Work with Public Key Certificates - An Experimental Study of OpenSSL Usability. In CT-RSA, volume 10808 of *Lecture Notes in Computer Science*, pages 45-64. Springer. https://doi.org/10.1007/978-3-319-76953-0_3

Weber, S., Coblenz, M., Myers, B., Aldrich, J., and Sunshine, J. (2017). Empirical studies on the security and usability impact of immutability. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 50-53. https://doi.org/10.1109/SecDev.2017.21

Werner, L., Hanks, B., and Mcdowell, C. (2004). Pair programming helps female computer science students. *ACM Journal of Educational Resources in Computing, 4*. https://doi.org/10.1145/1060071.1060075

Williams, L., Kessler, R. R., Cunningham, W., and Jeffries, R. (2000). Strengthening the case for pair programming. *IEEE Software, 17*(4):19-25. https://doi.org/10.1109/52.854064

Williams, L. and Upchurch, R. (2001). In support of student pair-programming. In *ACM SIGCSE Bulletin,(33*), pages 327-331. https://doi.org/10.1145/366413.364614

Williams, L., Wiebe, E., Yang, K., Ferzli, M., and Miller, C. (2002). In support of pair programming in the introductory computer science course. *Computer Science Education, 12*(3):197-212. https://doi.org/10.1076/csed.12.3.197.8618

Zeier, A., Wiesmaier, A., and Heinemann, A. (2019). *API Usability of Stateful Signature Schemes*, pages 221-240. Springer. https://doi.org/10.1007/978-3-030-26834-3_13