Write, Read, or Fix?
Exploring Alternative Methods for Secure Development Studies

Kelsey R. Fulton[*♡], Joseph Lewis[◇], Nathan Malkin[†♡], Michelle L. Mazurek[◇]
[*]*Colorado School of Mines;* [◇]*University of Maryland;* [†]*New Jersey Institute of Technology*

Abstract

When studying how software developers perform security tasks, researchers often ask participants to write code. These studies can be challenging because programming can be time-consuming and frustrating. This paper explores whether alternatives to code-writing can yield scientifically valid results while reducing participant stress. We conducted a remote study in which Python programmers completed two encryption tasks using an assigned library by either writing code from scratch, reading existing code and identifying issues, or fixing issues in existing code. We found that the read and fix conditions were less effective than the write condition in revealing security problems with APIs and their documentation, but still provided useful insights. Meanwhile, the read and especially fix conditions generally resulted in more positive participant experiences. Based on these findings, we make preliminary recommendations for how and when researchers might best use all three study design methods; we also recommend future work to further explore the uses and trade-offs of these approaches.

1 Introduction

Secure software development is a difficult task, as demonstrated by the many vulnerabilities discovered in production code on a regular basis [12, 29, 35]. Key causes of these vulnerabilities include developers failing to use the right tools or resources [4, 19, 31, 32, 42], making mistakes when writing code [52], or fundamentally misunderstanding necessary and important security concepts [19, 31, 52]. Addressing these challenges requires better understanding them; to wit, studying how developers approach secure development and how and why errors occur.

How best to conduct these studies, however, remains in some respects an open challenge. Many secure-development studies rely on code-writing tasks in order to observe developers' processes, decisions, and missteps [3,19,30,41,52]. These studies produce valuable results, but they can be very challenging to conduct: code-writing tasks are time-consuming and difficult to scope narrowly for lab experiments, and ecological validity is challenging because professional software development environments are hard to simulate [4, 40]. Further, the specialized population of software developers can be challenging to recruit and retain: they are difficult to reach, and they participate in studies outside their normal work hours, often for hourly rates much lower than their regular pay for software engineering [31, 45]. Frustration while participating in such studies can lead to high dropout rates, resulting in smaller, less powerful sample sizes [2].

Given this context, it is imperative for secure-development researchers to understand whether there are alternative approaches to conducting code writing studies that—at least for some research questions and contexts—will yield similar, scientifically valid results while reducing the stress and frustration for participants and researchers. As a first step to address this problem, Danilova et al. explored the possibility of substituting code review for code writing: participants wrote code reviews about snippets from a prior study about secure password storage [16]. The results indicated that code reviewing studies are potentially useful in addressing certain types of secure-development research questions.

Here, we build on this result with another exploratory study, this time investigating both reading and fixing insecure code as methodological alternatives to code writing. In contrast to Danilova et al., we directly compare these methods, measuring both secure-development outcomes and participants' experiences taking part in the study.

[♡]Work done while at University of Maryland

Copyright is held by the author/owner. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee.
USENIX Symposium on Usable Privacy and Security (SOUPS) 2024.
August 11–13, 2024, Philadelphia, PA, United States.

Specifically, we conducted a remote experimental study in which 112 Python programmers completed two symmetric encryption tasks (generating/storing a key and encrypting/decrypting data) using one of two encryption libraries. These study tasks were modeled on an earlier study by Acar et al. that measured the usability of various Python encryption APIs [2]. Participants were each assigned to one of three methodological conditions: writing secure code from scratch (*Write*), reading existing code and finding and explaining vulnerabilities and bugs (*Read*), or finding and fixing vulnerabilities and bugs in existing code (*Fix*). Participants then completed a survey about their experience taking part in the study.

Using this study, we address two key research questions:

RQ1 Do the *Read* and *Fix* conditions provide the same results about functionality and security as the *Write* condition?

RQ2 Do participants in the *Read* and *Fix* conditions experience fewer negative effects (drop-out rate, frustration, time to complete) than those in the *Write* condition?

We found that *Read* and *Fix* were less effective than *Write* in revealing security problems with APIs and their documentation. Participants in the *Fix* condition were particularly likely to focus only on getting code to run and pass provided tests; as such, they identified a much narrower set of vulnerabilities than *Read* participants. However, when they did identify vulnerabilities, *Fix* participants' attempts to remediate them did reveal interesting shortcomings in API documentation. Both the *Read* and *Fix* conditions provided insight into the kinds of vulnerabilities participants could recognize and remediate, in lower- and higher-effort scenarios respectively.

Overall, participants in the *Read* and *Fix* conditions reported more positive study experiences than *Write* participants, including more fun and less frustration. While participants in the *Read* condition spent slightly less time working on the study than those in the *Write* condition, participants in the *Fix* condition spent substantially less time.

Based on our qualitative and quantitative observations, we make preliminary recommendations for how and when researchers might best use all three study-design methods; we also recommend future work to further explore the uses and trade-offs of these approaches. Our exploratory study tests one possible implementation of *Write*, *Read*, and *Fix* study designs, in one experimental context aimed at answering a specific question about comparing APIs. As such, it cannot fully answer all questions about how and when it may be appropriate to deploy these methods. Nonetheless, we believe our study makes a valuable contribution to the ongoing evolution of best practices for empirical studies of secure development.

2 Related work

Prior research has explored how to recruit developers for studies and measure their efforts.

Participant recruitment. Prior work has studied the validity of varying recruitment approaches. Yamashita et al. explored the use of freelance marketplaces to recruit participants, concluding that while these services offer flexibility, low cost, and access to a wide population, there is uncertainty about developers' background and skills [53]. Baltes et al. evaluated the use of various sampling methodologies for software development studies, finding that sampling through public media yielded the best results [9]. Acar et al. explored the use of GitHub as a recruitment tool for studies by recruiting 307 active GitHub users to complete security-related programming tasks, finding no statistical difference for functionality or security between participants that were students and professionals [5]. To provide insight into the ecological validity of using computer science students in developer studies, Naiakshina et al. recruited professional software developers to complete programming problems from a prior study [31] and compared the results [30], finding that developers performed better than students and freelancers, but the treatment effects, such as prompting for security, held the same for developers and students. Similarly, Salman et al. discovered that students and professionals do not produce substantially different results in software studies [43].

Most recently, researchers have compared and contrasted a variety of recruitment venues, finding that crowdsourcing platforms require screening to get high-quality participants and that CS students at a variety of educational stages are a viable alternative for developers [25, 48]. To understand why professional developers may or may not participate in security studies, Serafini et al. conducted interviews with 30 developers and found that the length of the study, the topic of the study, compensation, and trust in the researchers had an impact on participation [45]. Interviewees were also concerned about their performance on security tasks.

Measuring developers' skill. Prior work has explored the construction of scales and survey questions to evaluate the skills of participants in software developer studies. Feigenspan et al. constructed a scale to measure programming experience of study participants by using questions from published studies that evaluated programming experience [17]. Comparing the participants' answers, they found self-efficacy to be an effective way to measure programming experience. Similarly, Bergersen et al. built a scale to measure programming experience by having 65 professional developers complete 19 Java programming exercises [10].

To aid in recruitment, Danilova et al. expanded this idea by building a screening questionnaire to help filter participants in programming studies, concluding with 6 recommended questions [15]. In follow-on work, they explored the use of time limits to increase the efficiency of screening questionnaires, concluding that implementing a time limit saves time and money while maintaining validity [13]. Focusing on measur-

ing security experience, Votipka et al. built a 15-item scale to measure the security self-efficacy of software developers [51].

Study design for secure development studies. Recent work has explored the use of a priori power calculations in the design of secure development studies, finding that many prior developer studies were underpowered to detect large effects [36].

Other work has explored the design of tasks provided in these studies. To explore the use of task deception, Naiakshina et al. had 40 students complete a password storage task, finding that priming participants for security had a statistically significant effect [33]. In a replication, Danilova et al. found that deception may not be necessary for ecological validity [14]. As an alternate approach to traditional in-person lab studies, Huaman et al. built a virtual study environment to allow researchers to conduct lab studies remotely, while maintaining ecological validity [24]. Most similar to our work, Danilova et al. evaluated the use of code review as a method for secure development studies, finding that code review could be a viable method for future developer studies, but recommending that more research be done into this alternative [16].

3 Method

To understand how *Read* and *Fix* compare to *Write*, we conducted a remote experimental study with Python programmers in which they completed two secure development tasks.

3.1 Study design

To ensure the validity of our study, we decided to partially replicate a prior study. This allowed us to compare the results of our *Write* with the original study's results, while also allowing us to compare the results of our *Write*, *Read*, and *Fix* conditions to each other. We chose to partially replicate a 2016 study from Acar et al. exploring the usability of various Python cryptography APIs [2] because it offered self-contained, short code writing tasks with results that lent themselves well to being compared across different conditions.

The original study [2] used five libraries and two types of encryption (symmetric and asymmetric encryption), for a total of 10 conditions. Comparing three different methods would triple this to 30 conditions, requiring a sample size that would not be feasible when recruiting and compensating developers. (The original study did not compensate participants.) As such, we opted to only replicate a subset of the original conditions (two libraries, one type of encryption), crossed with our three methods, for a total of six conditions.

We used the symmetric encryption tasks from the original study, as they served as the baseline for all of the regressions performed in the original paper, and participants in the symmetric condition produced more functional solutions. Participants were assigned to work in either PyCrypto or Cryptography.io. We selected PyCrypto as it was the baseline library for all the models in the original paper. We selected Cryptography.io as it proved to be significantly better in regards to security in the original paper and was designed with usability in mind. To facilitate replication, we used the versions of each library used in the original study, because many insecure defaults and security issues identified in the original study have been addressed in modern versions of the libraries. Finally, participants were assigned to one of three experimental conditions: writing code from scratch (*Write*); reading existing code, determining its correctness, and describing anything they wanted to change (*Read*); or reading existing code, determining its correctness, and fixing any vulnerabilities or bugs in the code (*Fix*).

Task selection. Participants were randomly assigned to one of the six conditions and tasked with completing an encrypt/decrypt task and a key generation and storage task. The order they were presented in was randomized.

For the *Write* condition, participants were given stub code and asked to write code that completed the described task. For the encrypt/decrypt task, this meant writing code that encrypted or decrypted plaintext or ciphertext, respectively, using a provided key. For the key generation and storage task, participants were tasked with creating an encryption key (file key) using a provided password, using the file key to encrypt and store another encryption key (task key) to a file in a provided directory name, and recovering the task key from the same file. They were provided with tests for each function and a set of cumulative tests at the very end. The tests covered the basic functionality of each task and ensured that the code ran without failure and returned the correct value. The key generation and storage function stub, encrypt/decrypt function stub, and provided tests for both tasks can be found in the Supplementary Materials.[1]

For the *Read* condition, participants were given already-completed code and asked to read it, determine its correctness, and add comments to identify what they would change and how. For the *Fix* condition, participants were given already-completed code and asked to read it, determine its correctness, and fix it if necessary. In both the *Read* and *Fix* conditions, the code provided contained four unique functionality bugs and one unique security vulnerability for the encrypt/decrypt task and two unique functionality bugs and four unique security vulnerabilities for the key generation and storage task. The functionality bugs ranged in type from those that would cause the code to crash when run to those that would not cause the provided tests to fail. The security vulnerabilities were based on vulnerabilities identified in the original paper [2], to best allow for comparing results between studies. A full list of the functionality bugs and security vulnerabilities in the provided code can be found in Table 1. Participants in the

[1]Supplementary materials, as well as a version of the paper with appendices included, can be found at https://osf.io/2nb3g/?view_only=8f83b46a6084440783d88c12e225a46c.

Task	Type	Vuln/bug	Description	Tests fail
Encrypt/decrypt	Security	Fixed IV	IV was set to static value	No
	Functionality	Return plaintext data	Return plaintext instead of ciphertext	Yes
	Functionality	Encrypt key instead of plaintext	Send key to encryption function instead of plaintext	Yes
	Functionality	Use plaintext as encryption key	Use plaintext as key to create cipher object	No
	Functionality	Return ciphertext data	Return ciphertext instead of plaintext	Yes
Key generation and storage	Security	Fixed salt	Salt was set to static value	No
	Security	Weak KDF	PBKDF1 used	No
	Security	Weak hash algorithm	Sha1 used	No
	Security	Bad mode selection	Insecure mode used to encrypt key	No
	Functionality	Wrong name to open	Send wrong variable to open command when writing key to file	Yes
	Functionality	Incorrect length for password check	Use wrong value to check length of password before using in keygen	No

Table 1: Vulnerabilities and bugs that were included in read and fix code snippets

Read and *Fix* conditions were provided with the same tests as participants in the write condition for reference (*Read*) or to test their code (*Fix*). All code and tests can be can be found in the Supplementary Materials.

Study environment. Our participants completed the study remotely through *NERDS* [26], a study environment based on the Developer Observatory [47]. *NERDS* is a customized Jupyter notebook environment that allows participants to complete coding tasks remotely, allowing for easier recruitment.

Once participants consented, they were taken to the instructions for the study. From there, they were able to begin participating. Participants in the *Write* and *Fix* conditions were able to run their code as often as they liked. Participants in the *Read* condition were not provided with a run button for their code, but they were, initially, still able to use hot keys to run code in Jupyter notebooks. These participants' solutions were removed from the study, and the interface was updated to remove this "feature." Participants were able to move forwards or backwards through the tasks at any point. Once participants completed all the tasks, they clicked a *finish* button that took them to the final survey. An example of our study infrastructure can be found in the Supplementary Materials.

Exit survey. Once participants indicated that they were finished with the tasks, they were directed to the final survey. We first asked participants about the perceived security and correctness of their solutions, how frustrating, fun, tedious, and challenging they found the tasks, and what was easy and hard about the tasks. The next section contained the Secure Software Development Self-efficacy Scale, which measures a person's perceived ability to complete various security tasks [51]. The survey concluded by asking participants about their security experience, general development experience, Python development experience, and demographics. Appendix A.1 contains the full survey.

3.2 Data analysis

We analyzed the data we collected using a mix of qualitative and quantitative analysis, which we describe below.

Qualitative analysis. To determine what vulnerabilities and bugs were introduced by participants in *Write* and not identified or fixed by participants in *Read* or *Fix*, we manually reviewed the submissions following processes from prior work [19, 52]. For the *Write* condition, two authors reviewed both tasks to identify any bugs or vulnerabilities present, using the vulnerabilities identified in the original Acar et al. study as a reference point. Each bug or vulnerability was labeled for type (functionality or security) and the specific vulnerability. In addition, we categorized the vulnerabilities into "issue" classes based on the classifications used in prior work [52].

For *Read* and *Fix*, the two authors reviewed submissions using a list of the bugs and vulnerabilities they knew to be present. For *Fix*, vulnerabilities or bugs included in the study setup were labeled for *both* whether participants were able to correctly identify and fix it. For the *Read* condition, vulnerabilities or bugs included in the study setup were labeled for whether participants were able to correctly identify the existence of the vulnerability/bug and whether the changes they proposed would fix the vulnerability/bug, how the participant said they would change the code to address the vulnerability/bug, and why. For both the *Read* and *Fix* conditions, we also labeled any additional issues identified by participants that were not actual bugs or vulnerabilities (false positives) and whether the unneeded fix introduced any new problems.

IRR was calculated for all variables using Krippendorff's α statistic, a conservative measure that considers coders' agreement as an improvement over randomly guessing. We met the recommended threshold for Krippendorff's α of 0.8 [22]. Prior to agreement, all vulnerabilities and bugs were confirmed by both coders, and consensus for all codes was reached through discussion. The final codebook and associated IRR values are in the Supplementary Materials.

Statistical comparisons. To compare our results among conditions and to the results from Acar et al., we performed logistic regressions to explore the impact of the library used on security and functionality (binary outcomes). We added a random intercept to account for multiple tasks from the same participant and used PyCrypto as the baseline, mirroring the regressions in Acar et al. [2]. To understand the impact of the condition on participants, we applied a linear regression for numeric outcomes (time spent), a poisson regression for count outcomes (number of vulnerabilities and bugs), and an ordinal logistic regression for Likert-scale outcomes (reported frustration/fun). For all regressions, the baseline was the *Write* condition to allow for better comparison of our new conditions to the more established experimental method.

3.3 Recruitment

We recruited from Upwork [1], an online freelancing platform, and computer science student mailing lists at multiple universities from May 2022 to July 2023, following best practices [25, 48]. We opted for multiple recruitment approaches to maximize the number of participants from a traditionally challenging population. Given that students and freelancers offer comparable conclusions [25, 30, 31, 48] and the original Github recruitment approach is no longer available, this mixed population proved effective.

Upwork allows researchers to filter participants based on their skillset. We filtered participants for experience with Python and age (18 or older). For the full study, participants were invited if, from their profile, they had completed at least one small project in Python. For recruitment from CS student mailing lists, we created a short screening survey to ensure that participants in the main study would have programming and Python experience. We started with a few questions to understand their general programming experience and occupation and concluded by using the questionnaire created by Danilova et al. [15] to determine if a participant actually had programming experience. A full copy of the screening survey can be found in Appendix A.2. We exclusively invited participants who had Python experience and were able to correctly answer all of the questions from the Danilova et al. questionnaire. Only 23/188 pre-screened participants failed the Danilova measure.

Participants were not compensated for the screening survey, but all main study participants were compensated $35 for completing the study with a possibility for a $5 bonus if participants were able to identify a majority (75%) of the functionality bugs present in the code. This bonus was meant to encourage people to give their best effort when participating in the study and focus on finding all issues within the code, rather than just the obvious issues. We framed the bonus as a reward for meeting a high correctness threshold (without specifying a number of bugs and only counting functionality bugs). Many participants (68% total, 58% of *Fix*, 81% of *Read*, and 94% of *Write*) received this bonus. While this may have impacted participant interaction, we deemed this important to promoting ecological validity. We discuss this further in Section 5. We discarded any responses where participants skipped all tasks, but kept responses that did not receive the bonus but where some attempt was made, as this gives us valuable insights into participant behavior.

3.4 Ethics and consent

Both surveys (pre-screen and final) and the full study were approved by University of Maryland's and Colorado School of Mines' Institutional Review Board. We obtained informed consent before the pre-screen survey and again before the full study. Participants were informed that they could skip any task or question and drop out of the study at any time.

3.5 Limitations

A key limitation of our work is the age of the study that we are replicating. The cryptographic libraries and their documentation have changed drastically since the original study. This means that any search of online materials for assistance would likely result in information that does not match the version of the library used in our study. We provided participants with a version of the documentation that matched the versions of libraries they were using, and we encouraged them to use the documentation as much as possible. Since we are not actually concerned with evaluating the current usability or security of the libraries, but rather with understanding the effect of the method used to study them, this limitation does not reduce the validity of our results.

The original study aimed to understand the usability of cryptography APIs for professional developers. We aim to semi-replicate this study. However, about half of our participants for this study were students, differing from the original study which used GitHub developers (a practice that is no longer allowed). While students often have less experience than professionals, several recent studies conclude they can be adequate substitutes in secure software development studies [25, 30, 31, 48], as many skilled professionals have limited experience with security specifically [3, 6, 11, 21, 23, 28, 42, 46, 52].

Our goal in this study was to understand the feasibility of using reading and/or fixing code as experimental substitutes for writing code in secure software development studies. This study serves as a single data point in this exploration. We explore whether reading and/or fixing code works to compare the usability of cryptography APIs. Additionally, the tasks in this study were deliberately small and self-contained to allow for easy comparison among the experimental conditions. Thus, our results may not generalize to all kinds of secure software development studies, such as those exploring other secure development issues or tasking participants with build-

	Upwork	CS mailing list
Total participants	*n*=76	*n*=36
Programming experience	7.4 years	5.7 years
Python experience	4.5 years	3 years
Professional programming experience	4.2 years	2.4 years
Professional Python experience	2.5 years	0.8 years
Security experience	1.4 years	0.8 years
Above-average security knowledge	68%	72%
SSD-SES total	46.3	40.8
SSD-SES Vulnerability	26.5	23.6
SSD-SES Communication	19.7	17.2

Table 2: Participant demographics

ing larger projects. However, we believe this study is a good first step toward exploring this phenomenon.

A final possible limitation of this work is the reliance on some self-report data to measure negative experiences of participants, such as reported frustration or fun. While self-report data can be biased or inaccurate, this is the best proxy we have for measuring frustration levels. We attempt to mitigate some of this self-report bias by also collecting other measures of negative effects, such as time spent or dropout rates, and consider these together as a measure for the negative experiences of our participants.

4 Results

In this section, we discuss our participants, our *Write* condition results as compared to the results from Acar et al. [2], and our *Read* and *Fix* compared to our *Write* condition results.

4.1 Participants

In total, 141 participants started our study, with 41 in the *Write* condition, 54 in the *Read* condition, and 46 assigned to the *Fix* condition. In total, 127 participants completed our study with 35 in *Write*, 48 in *Read*, and 44 in *Fix*. However, we removed 15 participants for a variety of reasons such as running or editing code in the *Read* condition prior to removing the use of hot keys ($N = 9$), skipping every task in the study ($N = 5$), and not understanding what to do ($N = 1$). This left us with 112 participants who completed the study; 35 in the *Write* condition, 37 in the *Read* condition, and 40 in the *Fix* condition. Details of participants assigned to each experimental condition and library can be found in Table 3.

Demographics. In general, our participants trended heavily toward male (80%), young (with ages ranging from 18 to 43 and 91% of participants being younger than 40), and educated (65% had at least a bachelor's degree). Our participants came from a variety of ethnic backgrounds, with a plurality identifying as Asian (48%).

On average, our participants had 6.8 years of programming experience and 4 years of Python experience. About 61% of our participants were employed in a professional role that required programming, with the most common job roles being developer and engineer. Of that 61%, 59% used Python in their job. Our participants had 3.8 years of professional programming experience and 2.1 years of professional Python experience. Finally, our participants had fairly little security experience, with an average of 1.2 years of security experience. However, our participants were self-confident in their security abilities, with 72% rating their security knowledge as at least average. While the experience of our participants may seem unusually high for a population including students, among education levels reported by final CS mailing list participants, 20 were consistent with being in college and 13 with being alumni or grad students (N = 36). At the institutions we recruited from, many undergrads enroll with significant high-school programming experience, so the high years of experience for our participants are not all that surprising. Demographics for each recruitment venue can be found in Table 2.

4.2 Replicating results from Acar et al.

First, we compare the *Write* condition to the functionality and security results from Acar et al. [2] (see Table 3).

Functionality. We considered participants' solutions to be functional if they ran, passed the provided tests, and completed the assigned task. If a participant skipped a task, the result was considered not functional. In Acar et al., participants were able to generate slightly more functional solutions using Cryptography.io than with PyCrypto, though this result was not statistically significant. In our study, slightly more PyCrypto participants than Cryptography.io participants produced functional solutions; this difference was likewise not statistically significant (Table 4).

Security.

For solutions deemed functional, we examined their security. In Acar et al., participants were able to generate significantly more secure solutions using Cryptography.io than PyCrypto. We similarly see significantly more secure solutions from participants using Cryptography.io than PyCrypto in our study (Table 3). Participants using Cryptography.io were 4.7× more likely to generate a secure solution (Table 4).

Comparing the security between the encrypt/decrypt and key generation and storage tasks, participants in the original study were most likely to produce a secure solution for the encrypt/decrypt task. We see a similar trend in our study (Table 3), although not as pronounced, with participants producing more secure solutions for the encrypt/decrypt task than the key generation and storage task (17 vs 10 solutions).

The distribution and types of vulnerabilities we found also followed the original study closely (see Table 5). In the en-

	Write			Read			Fix			Acar et al. [2]		
	P[1]	C[2]	T[3]	P	C	T	P	C	T	P	C	T
Started	21	20	***41***	26	28	***54***	22	24	***46***	136	136	***272***
Completed	18	17	***35***	24	24	***48***	21	23	***44***	48	48	***96***
Valid	18	17	***35***	19	18	***37***	21	19	***40***	41	39	***80***
Functionality	83%	59%	–	47%	31%	–	60%	84%	–	85%	90%	–
Key gen/storage	16	9	***25***	6	6	***12***	16	17	***33***	80%	80%	–
Encrypt/decrypt	14	11	***25***	12	5	***17***	9	15	***24***	90%	98%	–
Security	43%	70%	–	50%	36%	–	32%	31%	–	15%	70%	–
Key gen/storage	6	4	***10***	3	2	***5***	3	4	***7***	5%	30%	–
Encrypt/decrypt	7	10	***17***	6	2	***8***	5	6	***11***	20%	100%	–
Time (mins)	–	–	***38.2***	–	–	***30.2***	–	–	***22.5***	–	–	–

[1] PyCrypto [2]Crypto.io [3]Total

Table 3: Number of participants, across various conditions of interest. Percentages represent the share of functional/secure solutions among all/functional solutions. We report these percentages for consistency with the original paper.

		Write			Read			Fix		
Regression	**Factor**	**O.R.**	**C.I.**	***p***	**O.R.**	**C.I.**	***p***	**O.R.**	**C.I.**	***p***
Functionality	Cryptography.io	0.1	[0.0, 1.2]	0.082	0.5	[0.1, 1.7]	0.261	4.2	[1.3, 24.8]	0.034*
Security	Cryptography.io	4.7	[1.4, 19.4]	0.017*	3.0	[0.2, 55.9]	0.954	0.4	[0.0, 4.3]	0.430

Table 4: Final logistic regression for effect of library on functionality and security in each condition.

crypt/decrypt task, they matched exactly: the most common vulnerability was using a static initialization vector, followed closely by using a weak encryption mode and using a weak encryption algorithm. There were only slight differences between the studies in the key generation and storage tasks: in ours, the most common vulnerability was storing the key unencrypted, followed closely by failing to use a key derivation function (KDF), using a custom key derivation function, and using a static salt. In the original, the top three included using an insecure encryption mode (instead of the custom KDF).

4.3 Comparing functionality among conditions

Next, we compare the overall functionality results and the specific functionality bugs introduced, identified, and fixed, among our three conditions. Throughout this section, we use B_w to represent the number of bugs from the *Write* condition and, analogously, B_r for *Read*, and B_f for *Fix*.

Similar to *Write*, we considered a solution in the *Fix* condition to be functional if it ran, passed all the tests, and completed the task. For the the *Read* condition, we considered the code to be functional if the participant identified and correctly addressed all the functionality bugs we introduced, as described in Table 1. Specific counts for functional solutions per condition and task can be found in Table 3.

Overall, participants in *Write* and *Fix* were able to produce more functional solutions than those in *Read* for both libraries. This is likely because participants in *Read* were unable to run and test their code, which made identifying bugs difficult.

Comparing the two libraries to each other, in the *Read* condition, participants using PyCrypto were able to produce more functional solutions than those using Cryptography.io (47% vs 31%). This mirrors the result for the *Write* condition, discussed in Section 4.2 above. Conversely, participants in the *Fix* condition were more likely to produce a functional solution using Cryptography.io than PyCrypto (84% to 60%). However, none of these differences in any condition were statistically significant (Table 4).

Looking at the individual tasks, *Read* participants produced more functional solutions for the encrypt/decrypt task than the key generation and storage task. This again mirrors the result for *Write* as well as the original study. Conversely, in the *Fix* condition, participants produced more functional solutions for the key generation and storage task. We hypothesize that this relates to the specific functionality issues we inserted in *Fix*: participants in this condition appeared to prioritize passing the provided tests, which did not flag the functionality bug we inserted into the encrypt/decrypt task.

Examining bugs in *Write*, *Read*, and *Fix*. In total, participants introduced 34 bugs in the *Write* condition, left 81/222 bugs unidentified in the *Read* condition, and left 70/240 bugs unidentified in the *Fix* condition. In the *Write* and *Read* condition, participants introduced or left more bugs when using Cryptography.io (B_w = 25, B_r = 49) than when using PyCrypto (B_w = 9, B_r = 32). In the *Fix* condition, participants using PyCrypto (B_f = 38) introduced substantially more bugs than those who used Cryptography.io (B_f = 19).

Participants focus on 'test-centric' bugs in *Write* and *Fix*.

			Write			Read			Fix			Acar et al. [2]		
Function	**Issue**	**Vuln**	**P[1]**	**C[2]**	**T = 59[3]**	**P**	**C**	**T = 110**	**P**	**C**	**T = 176**	**P**	**C**	**T**
Encrypt/decrypt	*Static value*	Static IV	5	0	5	13	6	19	14	16	30	29	0	29
		Total	5	0	5	13	6	19	14	16	30	29	0	29
	Weak choice	Insecure alg	3	1	4	–	–	–	–	–	–	17	0	17
		Insecure mode	4	0	4	–	–	–	–	–	–	23	0	23
		Total	7	1	8	–	–	–	–	–	–	40	0	40
	Total		12	1	13	13	6	19	14	16	30	69	0	69
Key gen/storage	*No encryption*	Key plain	3	7	10	–	–	–	–	–	–	4	7	11
		Total	3	7	10	–	–	–	–	–	–	4	7	11
	Static value	Static IV	2	0	2	–	–	–	–	–	–	3	0	3
		Static salt	2	3	5	11	9	20	19	17	36	1	10	11
		Static key	2	2	4	–	–	–	–	–	–	–	–	–
		Total	6	5	11	11	9	20	19	17	36	4	10	14
	Weak choice	No KDF	6	3	9	–	–	–	–	–	–	15	1	16
		Custom KDF	5	0	5	–	–	–	–	–	–	0	0	0
		Weak KDF	–	–	–	11	–	11	20	–	20	1	0	1
		Weak hash	–	–	–	–	9	9	–	17	17	0	0	0
		KDF iter	3	0	3	14	11	25	16	20	36	2	0	2
		Insecure alg	3	0	3	–	–	–	–	–	–	11	0	11
		Insecure mode	5	0	5	15	11	26	19	18	37	14	0	14
		Total	22	3	25	40	31	71	55	55	110	43	1	44
	Total		31	21	46	53	37	91	74	72	146	51	18	69

[1] PyCrypto [2]Crypto.io [3]Total

Table 5: Number of vulnerabilities for each issue and the number of projects each vulnerability was introduced in.

For most functionality bugs in the *Write* condition, participants' code did run and pass the provided tests; however, it did not complete the required task. For example, the most common functionality issue in *Write* was caused by failing to store the encryption information correctly ($B_w = 6$), such as failing to store the key in the provided directory, as per the instructions ($B_w = 4$), or not storing the key in a file ($B_w = 2$).

Similarly, most of the bugs unidentified in the *Fix* condition were caused by failing to complete the task rather than the code failing to run or pass the tests ('test-centric' bugs). The most common functionality issue left unidentified in the *Fix* condition was caused by inconsistent checking for password length in the key derivation function ($B_f = 30$). The second most common was failing to identify that unencrypted data was being returned from the encrypt function and encrypted data was being returned from the decrypt function ($B_f = 13$). This aligns with prior work showing that developers often assume that if their code runs and passes provided tests, then it is correct and secure [7, 19].

Participants identify a greater variety of bugs in *Read*. Conversely, in the *Read* condition, bugs that caused the code to not run or not pass the provided tests went unidentified as often as those that caused the code to fail to complete the task. The least identified in *Read* were cases where the wrong variable name was passed to or used in a function, causing the code to crash if run ($B_r = 44$). For example, about half of participants failed to identify a bug where a Python keyword was passed to the open function in Python ($B_r = 17$), causing a crash. Participants were equally unable to identify the inconsistent check for the password length ($B_r = 16$) and returning (un)encrypted data in the encrypt and decrypt functions ($B_r = 20$). This suggests that *Read* participants, unable to run code, review all the code equally closely, resulting in identifying fewer overall but more diverse functionality issues.

4.4 Comparing security among conditions

Next, we explore how often participants who produced a functional solution were also able to produce a secure solution. We compare across all three conditions, and then discuss in detail the vulnerabilities introduced, identified, and fixed in each. Throughout this section, we use V_w, V_r, and V_f to represent the number of vulnerabilities in the *Write* condition, the *Read* condition, and the *Fix* condition, respectively. For vulnerabilities that were unique to the *Write* condition, we do not include counts for V_r and V_f.

Overall, participants in the *Write* condition produced more secure solutions than those in *Read* and *Fix*. This is perhaps unsurprising, as *Write* participants started with a blank slate, rather than starting with vulnerabilities already included.

In the *Read* and *Fix* conditions, we find little to no difference when comparing the two libraries. *Read* and *Fix* participants produced slightly more secure solutions with PyCrypto

than Cryptography.io (50% to 36% and 32% to 31% respectively), but these comparisons were not statistically significant (Table 4). Importantly, this differs from the result in the *Write* condition (Section 4.2), which (like the original paper) identified Cryptography.io as meaningfully better for security.

In all three conditions, participants were able to produce more secure solutions for the encrypt/decrypt task than the key generation and storage task, mirroring the original study.

Examining vulnerabilities in *Write*, *Read*, and *Fix*. In total, participants introduced 59 vulnerabilities in the *Write* condition, left 110 vulnerabilities unidentified in the *Read* condition, and left 176 vulnerabilities unidentified in the *Fix* condition. The large disparity in vulnerabilities between the conditions was likely due to the fact that participants in *Read* and *Fix* started with vulnerabilities in their codebase due to the study setup, while participants in the the *Write* condition started with a blank slate.

Participants assigned to use PyCrypto (V_w = 43, V_r = 66, V_f = 88) introduced or left unidentified more vulnerabilities than those assigned to use Cryptography.io (V_w = 22, V_r = 43, V_f = 88) in *Write* and *Read* conditions. This aligns with the original study and is likely due to the relative simplicity of the Cryptography.io library as well as the several secure examples within its documentation. Table 5 shows counts for vulnerability types across conditions and libraries; for explanations of vulnerabilities we inserted into the read and fix conditions, refer back to Table 1.

Participants misunderstood cryptography implementations in all conditions, but documentation weaknesses are more visible in *Write*. In every condition, the most common type of vulnerability introduced or left unidentified involved participants attempting to implement cryptography protocols but making a weak cryptography choice (V_w = 33/59, V_r = 71/110 , V_f = 110/176). The second most common issue in all three conditions was using a fixed or static value where randomness is needed (V_w = 16, V_r = 39, V_f = 66).

In *Write*, both of these vulnerability types occurred overwhelmingly among participants using the PyCrypto library (V_w = 34 and 11 respectively) rather than Cryptography.io (V_w = 4 and 5 respectively), mirroring the original study. As Acar et al. note, this was likely caused by the structure of the then-current PyCrypto documentation, which made identifying and using the most secure options difficult.

Interestingly, we don't see the same pattern in the other two conditions, where vulnerabilities caused by a weak cryptography choice appear nearly equally in both libraries (Cryptography.io: V_r = 31, V_f = 55; PyCrypto: (V_r = 40, V_f = 55). Similarly, static value problems were fairly evenly distributed between the libraries in *Read*and *Fix*(Cryptography.io: V_r = 15, V_f = 33; PyCrypto (V_r = 24, V_f = 33). We hypothesize that this occurs because identifying vulnerabilities is a difficult task, regardless of the library used. This suggests that, as currently constructed, study designs using *Read* and *Fix* would not have identified a key problem in PyCrypto that was clearly evident in the original study. We hypothesize that better documentation and simpler APIs (as found in Cryptography.io at the time) have a larger effect when writing code, but are less salient when trying to identify pre-existing bugs, two very different processes.

Focus on testing in *Fix* causes vulnerabilities to be missed. Participants correctly identified 75 out of 185 vulnerabilities in the *Read* condition but only 24 out of 200 vulnerabilities in the *Fix* condition. Our final Poisson regression model estimates that *Read* participants identified 1.65× more vulnerabilities than *Fix* participants ($p < 0.001$). As with functionality, we attribute this to *Fix* participants' extreme prioritization of passing the provided tests: every single *Fix* participant started the study by running the code first, and 31 of 40 moved on immediately as soon as the code ran successfully. As noted above, this aligns with prior work regarding developers' assumptions that runnable code is correct [8, 19].

Interestingly, participants identified 8 items in the *Read* condition but only 1 item in the *Fix* condition that were non-vulnerabilities. Some of these were valid security-relevant issues outside the scope of the assigned tasks (e.g., including integrity checks for the encrypted data). Others reflected conceptual misunderstandings (e.g., two *Read* participants flagged that encryption was missing an initialization vector, but failed to notice that the code used ECB mode, which does not require an initialization vector but is highly insecure). This result also suggests that *Read* participants paid closer attention to details (even when getting some of them wrong), most likely because they could not run the code to get feedback.

Once vulnerabilities are found, *Read* and *Fix* participants face similar challenges remediating them. Not only did *Read* participants identify more vulnerabilities than *Fix* participants, they were also better at successfully remediating vulnerabilities (V_r = 61/75 vs. V_f = 12/24 , respectively). However, they tended to struggle with remediating similar issues. In one notable example, 10 participants in the *Read* condition and 3 participants in the *Fix* condition noticed that the provided code used insufficient iterations in the key derivation function. The recommended value (at the time of the original study) is 10,000, but several participants (V_r = 4, V_f = 1) increased the value to only 1000, as recommended by the PyCrypto documentation. Only three of the 22 *Write* participants who used a key derivation function failed to use at least 10,000 iterations. Here, the *Read* and *Fix* study designs are able to illuminate a problem in the library documentation.

4.5 Effects on participants

In this section, we discuss the effect of the different experimental conditions on participants and response quality, including dropout rate, time to complete, and reported frustration and fun. Throughout this section we use N_r, N_w, and N_f to

Factor	Coeff	C.I.	p
Read	-7.3	[-22.2, 7.6]	0.341
Fix	-15.7	[-30.4, -1.1]	0.037*

Table 6: Final linear regression for completion time.

represent the number of participants in *Read*, *Write*, and the *Fix* condition, respectively.

Dropouts. Overall, only 11 of 141 participants dropped out, (N_w = 6/41, N_r = 3/54, N_f = 2/46), which is drastically different than the experience of the researchers in Acar et al. [2], who had a dropout rate of nearly 84%. This is likely due primarily to the fact that we compensated participants, but only if they completed the study, thus incentivizing them to finish. (Another contributing factor is likely that we did not include the KeyCzar and M2Crypto libraries or asymmetric encryption tasks, all of which were associated with especially high dropout rates in the original study.) Details of the number of valid and completed submissions can be seen in Table 3.

Completion time. We use the time spent on each condition as a first, crude measure of stress on participants, as participants consider time as an important factor when enrolling in secure software development studies [45]. To calculate this, we measured the time spent *actively* working on the study, i.e. excluding any breaks, by measuring the amount of time participants had our study actively open on their screen. The mean completion time for the study as a whole was 30.2 minutes (η = 19.6 minutes, σ = 32.6 minutes). Participants spent 38.2 minutes on average in the *Write* condition (η = 19.9 minutes, σ = 43.7 minutes), 30.9 minutes on average in the *Read* condition (η = 22.3 minutes, σ = 25.7 minutes), and 22.5 minutes on average in the *Fix* condition (η = 16.6 minutes, σ = 25.1 minutes). Using a linear regression, we found that participants in the *Fix* condition spent significantly less time than those in the *Write* condition ($p = 0.037$, $CI = [-30.37, -1.11]$). Details of this regression can be found in Table 6. We note that while not substantially different across conditions, we do see the fewest dropouts in *Fix*, the condition with the shortest completion time.

***Fix* reported as least frustrating, *Write* as most frustrating.** To measure study stress on participants, we asked them to self-report their frustration with the required tasks. We used an ordinal logistic regression for each task (encrypt/decrypt and key generation and storage) to understand the impact of condition/library on frustration as described in Section 3.2.

For the encrypt/decrypt task, 10 participants (29%) in the *Write* condition, 10 participants (27%) in the *Read* condition, and 6 participants (15%) in the *Fix* condition reported being frustrated (agree and strongly agree). Our model estimates that participants in the *Fix* condition were 2.71× more likely to report lower frustration than those in the *Write* condition (p =

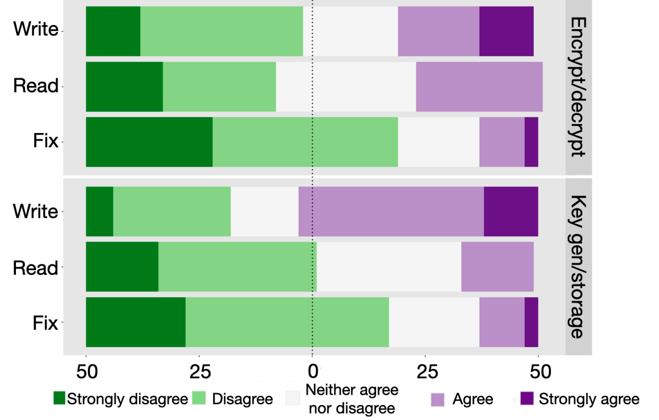

Figure 1: Reported frustration for both tasks in each condition

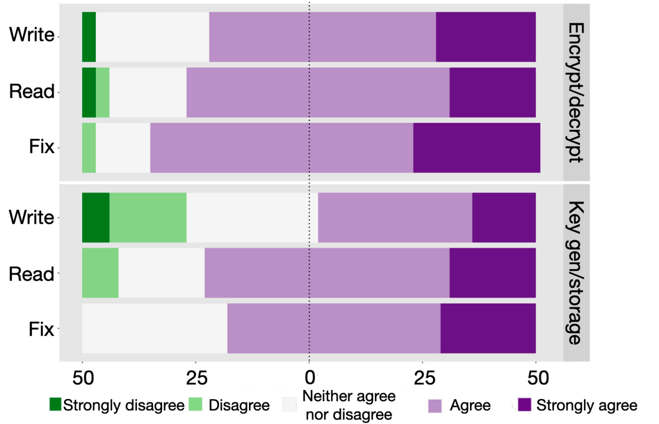

Figure 2: Reported fun for both tasks in each condition

0.02). For the key generation and storage task, 18 participants (51%) in the *Write* condition, 6 (17%) in the *Read* condition, and 4 participants (10%) in the *Fix* condition reported being frustrated (agree and strongly agree). Our model estimates that participants in the *Read* condition and the *Fix* condition were 3.7 and 6.0× more likely to report lower frustration, respectively, than those in the *Write* condition ($p = 0.004$, $p < 0.001$ respectively). We report the model and p-values in Table 7. We found no significant effect of the library on reported frustration for either task. The reported frustration in each condition for both tasks can be seen in Figure 1.

***Read* and *Fix* reported as more fun than *Write* for key generation and storage.** As another measure of study stress on participants, we asked participants to self-report whether they found the tasks fun. For the encrypt/decrypt task, 23 participants (66%) in *Write*, 28 participants (76%) in *Read*, and 34 participants (85%) in *Fix* reported having fun with the tasks (agree and strongly agree). To understand the effect of the condition and library on fun, we used the same ordinal logistic regression as described above. We found no significant effect of the condition or library on reported fun. For the

Regression	Factor	Enc/dec O.R.	Enc/dec C.I.	Enc/dec p	Keygen O.R.	Keygen C.I.	Keygen p
Frustration	Read	1.1	[0.5, 2.6]	0.834	3.7	[1.5, 9.1]	0.004*
	Fix	2.7	[1.2, 6.5]	0.023*	6.0	[2.5, 15.1]	<0.001*
	Crypto.io	0.8	[0.4, 1.7]	0.682	1.1	[0.5, 2.1]	0.887
Fun	Read	1.1	[0.4, 2.7]	0.862	2.6	[1.1, 6.4]	0.033*
	Fix	1.7	[0.7, 4.3]	0.246	2.7	[1.1, 6.5]	0.028*
	Crypto.io	1.1	[0.5, 2.2]	0.889	1.0	[0.5, 1.9]	0.909

Table 7: Ordinal logistic regression for frustration and fun.

key generation and storage task, 17 participants (49%) in the *Write* condition, 27 participants (73%) in the *Read* condition, and 36 participants (90%) in the *Fix* condition reported having fun with the tasks (agree and strongly agree). Our model estimates that participants in the *Read* and *Fix* conditions were each 2.6× and 2.7× more likely to report agreeing with having fun than those in the *Write* condition, respectively. We saw no significance of the library on reported fun. We report the model and p-values in Table 7. The reported fun in each condition for both tasks can be seen in Figure 2.

These results suggest that participants overall felt more positive about *Read* and *Fix* than *Write*, including actually enjoying *Read* and *Fix* for some tasks, with a minority of participants reporting frustration in *Fix* and a majority of participants reporting having fun in *Read* and *Fix*. A more positive participant experience has a number of potential benefits, including increased effort and better retention [45].

5 Discussion and recommendations

We now discuss how the research community can potentially apply our findings. We note that our results are exploratory in nature and that further work is likely needed to validate these findings, particularly in new contexts; we discuss this need for additional study further in Section 5.2.

5.1 When to use *Write*, *Read*, or *Fix*

Based on our experience, we make preliminary recommendations for how to use the three different study designs in future research, subject to further validation.

Use *Write* for measuring the efficacy of code writing tools. Similar to other studies that have relied on participants writing code [2, 19, 30, 41, 42, 52], our results in *Write* were able to reveal important differences between the cryptographic APIs that we tested, namely in the security of the solutions participants produced. These differences were substantially less visible in the *Read* and *Fix* conditions. We hypothesize that this may be because "simplified" cryptographic APIs are designed to prevent developers from making, rather than identifying and fixing security mistakes. Cryptographic APIs and documentation often contain examples supporting code-writing, but these are non-exhaustive and usually don't document incorrect usage, forcing users to reference other resources when reviewing code. The *Write* method may be more appropriate when the researchers' goal is to evaluate tools aimed at *writing* secure code, such as APIs [2, 37, 50], IDE tools [18, 20, 27, 34, 41, 49], and AI-based code generation assistants [38, 39, 44], or when the goal is to categorize the security results of building a system from scratch. In particular, *Write* would be ideal for a study exploring the types of vulnerabilities that developers introduce in specific programming tasks and languages [52].

Use *Read* to measure developers' knowledge. *Read* participants identified fewer functionality bugs but more vulnerabilities than *Fix* participants, despite having no provided tests and being unable to run the code. This suggests a *Read* study design may be useful to understand the types of bugs and especially vulnerabilities developers know to look for in a given development context. This could be useful for evaluating overall security awareness and knowledge, addressing research questions about how well developers can spot problems or achieve security when they are required to pay close attention to details, providing (in some ways) an upper bound on secure development skills. For example, *Read* could be useful for understanding whether developers know they need to salt passwords to store them securely [32]. Rather than having participants build an entire system from scratch, researchers could provide participants with finished code that did (not) salt passwords, and see whether participants can spot the problem, providing a much faster approach.

Use *Fix* to measure quick fixes. Participants in the *Fix* condition found more functionality bugs than security vulnerabilities, and overall they mostly caught 'test-centric' issues, rather than deeper or less visible issues. These results echo prior work, in which participants struggled to identify new areas for testing when any tests were provided [7, 19]. This suggests *Fix* study designs may be useful for identifying vulnerabilities and bugs developers are able to quickly recognize and address using existing tests and prior knowledge. These results may therefore serve as (in some ways) a lower bound on programmers' secure development abilities.

Further, the utility of *Fix* study designs could potentially be extended to studies of secure-development tools (e.g., static analysis tools or fuzzers) that automatically flag certain types of vulnerabilities for developers' attention or of how participants select and evaluate AI-generated code suggestions, as these studies highlight how the evaluation and interaction is performed rather than just the identification. We found that once issues are identified, *Read* and *Fix* participants had roughly similar success in addressing them, so this study design may be useful for examining how effectively developers can understand and address vulnerabilities these tools iden-

tify.

Use *Read* and *Fix* to minimize time, frustration. Compared with the the *Write* condition, participants who had to *Read* or *Fix* code spent, on average, less time completing the study and found their tasks less frustrating and more fun. Prior research has found that industry developers—a desirable but hard-to-reach demographic for secure development studies—prioritize factors such as study duration and low effort [45]. Therefore, when appropriate for the research questions, *Read* and *Fix* methods may offer reasonable trade-offs to researchers who are concerned about recruiting enough target participants for studies.

5.2 How to design *Write*, *Read*, *Fix* studies

Since the *Read* and *Fix* methods for secure development studies are relatively novel, there are several design considerations future researchers should take into account. In this preliminary study, we were only able to explore a few points in this potential design space; we hope that going forward, other researchers will explore different trade-offs and design choices, to better characterize the pros and cons of different study designs in the broader secure-development context.

Improving *Write* studies. We suspect that there are limits to how enjoyable write studies can be designed to be. Our low-dropout experience, compared to Acar et al. [2], suggests compensation and clear study expectations help. Echoing prior work, our results also suggest that fun and time spent matters, so more interesting and shorter tasks may help [45].

Utilizing bonus payments. While bonus payments have not previously been widely used, we added them after observing some participants skipping all tasks to receive compensation. We specifically left the framing of the bonus vague so as to not sway participants too heavily to only address a certain number of bugs. The addition of the bonus improved the validity of our results, as it reduced the number of low-effort submissions. However, it is possible that it focused participants' attention on functionality bugs, as they were motivated to receive the bonus, possibly causing them to not look for security vulnerabilities. Researchers should consider the inclusion and framing of a bonus carefully. While it helps promote participant retention and quality, it may also sway participants in the *Fix* condition to focus solely on passing the tests, rather than taking a more holistic approach.

Inserting realistic vulnerabilities. In order to conduct a study based on reading or fixing code, researchers must insert appropriately realistic vulnerabilities (and potentially develop predefined tests or tools that can flag them). For this early-stage, exploratory work, we derived these vulnerabilities from real participant examples observed in a prior code-writing study. Of course, researchers who are using *Read* or *Fix* instead of a *Write* study are unlikely to have this kind of prior data available. We suggest instead identifying realistic security defects from known vulnerability listings (e.g., CVE lists) or using examples taken from open-source software or from programming sites like Stack Overflow. It may also be possible to derive defects from interview or survey studies that reveal developers' misconceptions and mental models. These approaches, however, may need to be validated with further studies of experimental methods.

Enhancing ecological validity. For this early-stage study comparing methods, we prioritized internal validity (straightforward comparisons between conditions) as well as replicating prior work (for contextualizing our results). This led to specific design choices, such as restricting participants to specific libraries, avoiding external documentation, using older versions of software and documentation, and preventing *Read* participants from running code. We believe these choices made sense for this study, but they did reduce ecological validity, as real-world code review processes obviously lack many or all of these restrictions. Researchers considering employing *Read* and *Fix* should revisit these trade-offs for their own research questions and constraints; for example, in some studies it may be useful to more closely match the flow of real-world code review processes such as those used in public repositories like GitHub.

One of the main reasons *Fix* underperformed was because participants focused primarily on passing the pre-written tests; since these are not an inherent feature of *Fix*, future work should investigate whether removing them would increase the overall effectiveness of this condition. As with other exploratory changes, versions of *Read* and *Fix* that prioritize ecological validity should be validated with additional methodological studies.

Exploring *Read* and *Fix* for other secure-development domains and questions. We designed our study around a specific secure-development research question: the relative effectiveness of cryptographic APIs. Other security domains may suffer from different types of vulnerabilities and bugs, and other types of research questions may exhibit different outcomes from different study designs. Further work is needed to explore how *Read* and *Fix* would function in these different contexts. Perhaps other novel methods, in addition to the three study designs we considered, can also be developed to address these contexts.

6 Acknowledgments

We thank the anonymous reviewers who provided helpful comments on this paper. This project was supported by NSF grants CNS-1801545 and CAREER-1943215.

References

[1] Upwork. https://www.upwork.com/.

[2] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. Comparing the Usability of Cryptographic APIs. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017.

[3] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. You Get Where You're Looking For: The Impact of Information Sources on Code Security. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016.

[4] Yasemin Acar, Sascha Fahl, and Michelle L Mazurek. You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users. In *IEEE Cybersecurity Development (SecDev)*. IEEE, 2016.

[5] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L Mazurek, and Sascha Fahl. Security Developer Studies with GitHub Users: Exploring a Convenience Sample. In *Symposium on Usable Privacy and Security (SOUPS)*, 2017.

[6] Noura Alomar, Primal Wijesekera, Edward Qiu, and Serge Egelman. "You've Got Your Nice List of Bugs, Now What?" Vulnerability Discovery and Management Processes in the Wild. In *Symposium on Usable Privacy and Security (SOUPS)*, 2020.

[7] Hala Assal and Sonia Chiasson. Security in the Software Development Lifecycle. In *Symposium on Usable Privacy and Security (SOUPS)*, 2018.

[8] Wei Bai, Omer Akgul, and Michelle L Mazurek. A Qualitative Investigation of Insecure Code Propagation from Online Forums. In *IEEE Cybersecurity Development (SecDev)*. IEEE, 2019.

[9] Sebastian Baltes and Stephan Diehl. Worse Than Spam: Issues in Sampling Software Developers. In *ACM/IEEE international symposium on empirical software engineering and measurement*, 2016.

[10] Gunnar R Bergersen, Dag IK Sjøberg, and Tore Dybå. Construction and validation of an instrument for measuring programming skill. *IEEE Transactions on Software Engineering*, 40(12), 2014.

[11] Veroniek Binkhorst, Tobias Fiebig, Katharina Krombholz, Wolter Pieters, and Katsiaryna Labunets. Security at the End of the Tunnel: The Anatomy of VPN Mental Models Among Experts and Non-Experts in a Corporate Context. In *USENIX Security Symposium (USENIX Security)*, 2022.

[12] Yung-Yu Chang, Pavol Zavarsky, Ron Ruhl, and Dale Lindskog. Trend Analysis of the CVE for Software Vulnerability Management. In *IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*. IEEE, 2011.

[13] Anastasia Danilova, Stefan Horstmann, Matthew Smith, and Alena Naiakshina. Testing Time Limits in Screener Questions for Online Surveys with Programmers. In *IEEE International Conference on Software Engineering (ICSE)*, 2022.

[14] Anastasia Danilova, Alena Naiakshina, Johanna Deuter, and Matthew Smith. Replication: On the Ecological Validity of Online Security Developer Studies: Exploring Deception in a Password-Storage Study with Freelancers. 2020.

[15] Anastasia Danilova, Alena Naiakshina, Stefan Horstmann, and Matthew Smith. Do you really code? Designing and Evaluating Screening Questions for Online Surveys with Programmers. In *International Conference on Software Engineering (ICSE)*. IEEE, 2021.

[16] Anastasia Danilova, Alena Naiakshina, Anna Rasgauski, and Matthew Smith. Code Reviewing as Methodology for Online Security Studies with Developers-A Case Study with Freelancers on Password Storage. In *Symposium on Usable Privacy and Security (SOUPS)*, 2021.

[17] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring programming experience. In *IEEE international conference on program comprehension (ICPC)*. IEEE, 2012.

[18] Matthew Finifter and David Wagner. Exploring the Relationship Between Web Application Development Tools and Security. 2011.

[19] Kelsey R Fulton, Daniel Votipka, Desiree Abrokwa, Michelle L Mazurek, Michael Hicks, and James Parker. Understanding the How and the Why: Exploring Secure Development Practices through a Course Competition. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[20] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl. Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse. In *Symposium on Usable Privacy and Security (SOUPS)*, 2018.

[21] Matthew Green and Matthew Smith. Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security & Privacy*, 14(5), 2016.

[22] Andrew F Hayes and Klaus Krippendorff. Answering the Call for a Standard Reliability Measure for Coding Data. *Communication methods and measures*, 1(1).

[23] Mohammadreza Hazhirpasand, Oscar Nierstrasz, Mohammadhossein Shabani, and Mohammad Ghafari. Hurdles for Developers in Cryptography. 2021.

[24] Nicolas Huaman, Alexander Krause, Dominik Wermke, Jan H. Klemmer, Christian Stransky, Yasemin Acar, and Sascha Fahl. If You Can't Get Them to the Lab: Evaluating a Virtual Study Environment with Security Information Workers. In *Symposium on Usable Privacy and Security (SOUPS)*, 2022.

[25] Harjot Kaur, Sabrina Amft, Daniel Votipka, Yasemin Acar, and Sascha Fahl. Where to Recruit for Security Development Studies: Comparing Six Software Developer Samples. 2022.

[26] Joe Lewis. Nerds. https://github.com/SP2-MC2/Developer-Observatory, 2021.

[27] Tianshi Li, Yuvraj Agarwal, and Jason I Hong. Coconut: An IDE Plugin for Developing Privacy-Friendly Apps. *ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(4), 2018.

[28] Abraham H Mhaidli, Yixin Zou, and Florian Schaub. "We Can't Live Without Them!" App Developers' Adoption of Ad Networks and Their Considerations of Consumer Risks. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*, pages 225–244, 2019.

[29] Mitre. CVE. https://cve.mitre.org/, 2020.

[30] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, and Matthew Smith. On Conducting Security Developer Studies with CS Students: Examining a Password-Storage Study with CS Students, Freelancers, and Company Developers. In *CHI Conference on Human Factors in Computing Systems*, 2020.

[31] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, Emanuel Von Zezschwitz, and Matthew Smith. "If you want, I can store the encrypted password": A Password-Storage Field Study with Freelance Developers. In *CHI Conference on Human Factors in Computing Systems*, 2019.

[32] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. Why Do Developers get Password Storage Wrong? A Qualitative Usability Study. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[33] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith. Deception Task Design in Developer Password Studies: Exploring a Student Sample. In *Symposium on Usable Privacy and Security (SOUPS)*, 2018.

[34] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[35] NIST. National Vulnerability Database. https://nvd.nist.gov/general, 2020.

[36] Anna-Marie Ortloff, Christian Tiefenau, and Matthew Smith. SoK: I Have the (Developer) Power! Sample Size Estimation for Fisher's Exact, Chi-Squared, McNemar's, Wilcoxon Rank-Sum, Wilcoxon Signed-Rank and t-tests in Developer-Centered Usable Security. In *Symposium on Usable Privacy and Security (SOUPS)*, 2023.

[37] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. Usability Smells: An Analysis of Developers' Struggle With Crypto Libraries. 2019.

[38] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022.

[39] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do Users Write More Insecure Code with AI Assistants? In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[40] Olgierd Pieczul, Simon Foley, and Mary Ellen Zurko. Developer-centered security and the symmetry of ignorance. In *New Security Paradigms Workshop*, 2017.

[41] Stephan Plöger, Mischa Meier, and Matthew Smith. A Qualitative Usability Evaluation of the Clang Static Analyzer and libFuzzer with CS Students and CTF Players. In *Symposium on Usable Privacy and Security (SOUPS)*, 2021.

[42] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L Mazurek, and Piotr Mardziel. Build It, Break It, Fix It: Contesting Secure Development. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[43] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. Are Students Representatives of Professionals in Software Engineering Experiments? In *IEEE International Conference on Software Engineering (ICSE)*, 2015.

[44] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. In *USENIX Security Symposium (USENIX Security)*, 2023.

[45] Raphael Serafini, Marco Gutfleisch, Stefan Albert Horstmann, and Alena Naiakshina. On the Recruitment of Company Developers for Security Studies: Results from a Qualitative Interview Study. In *Symposium on Usable Privacy and Security (SOUPS)*, 2023.

[46] Matthew Smith. Usable Security—The Source Awakens. USENIX Association, 2016.

[47] Christian Stransky, Yasemin Acar, Duc Cuong Nguyen, Dominik Wermke, Doowon Kim, Elissa M. Redmiles, Michael Backes, Simson Garfinkel, Michelle L. Mazurek, and Sascha Fahl. Lessons Learned from Using an Online Platform to Conduct Large-Scale, Online Controlled Security Experiments with Software Developers. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*. USENIX Association, 2017.

[48] Mohammad Tahaei and Kami Vaniea. Recruiting Participants With Programming Skills: A Comparison of Four Crowdsourcing Platforms and a CS Student Mailing List. In *CHI Conference on Human Factors in Computing Systems*, 2022.

[49] Mohammad Tahaei, Kami Vaniea, Konstantin Beznosov, and Maria K Wolters. Security Notifications in Static Analysis Tools: Developers' Attitudes, Comprehension, and Ability to Act on Them. In *CHI Conference on Human Factors in Computing Systems*, 2021.

[50] Christian Tiefenau, Emanuel von Zezschwitz, Maximilian Häring, Katharina Krombholz, and Matthew Smith. A Usability Evaluation of Let's Encrypt and Certbot: Usable Security Done Right. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1971–1988, 2019.

[51] Daniel Votipka, Desiree Abrokwa, and Michelle L Mazurek. Building and Validating a Scale for Secure Software Development Self-Efficacy. In *CHI Conference on Human Factors in Computing Systems*, 2020.

[52] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It. In *USENIX Security Symposium (USENIX Security)*, 2020.

[53] Aiko Yamashita and Leon Moonen. Surveying developer knowledge and interest in code smells through online freelance marketplaces. In *International Workshop on User Evaluations for Software Engineering Researchers (USER)*. IEEE, 2013.

A Survey instruments

A.1 Final survey

A.1.1 Encrypt/decrypt task specific questions

1. Recall your experiences with the task where you were expected to encrypt/decrypt data (**encryption/decryption task**).
2. I completed the **encryption/decryption task** correctly.
   - I am not confident.
   - I am slightly confident.
   - I am somewhat confident.
   - I am moderately confident.
   - I am absolutely confident.
3. I completed the **encryption/decryption task** securely.
   - I am not confident.
   - I am slightly confident.
   - I am somewhat confident.
   - I am moderately confident.
   - I am absolutely confident.
4. The documentation was helpful in completing the **encryption/decryption task**.
   - Strongly agree
   - Agree
   - Neither agree nor disagree
   - Disagree
   - Strongly disagree
5. Completing the **encryption/decryption task** was frustrating.
   - Strongly agree
   - Agree
   - Neither agree nor disagree
   - Disagree
   - Strongly disagree
6. Completing the **encryption/decryption task** was fun.
   - Strongly agree
   - Agree
   - Neither agree nor disagree
   - Disagree
   - Strongly disagree
7. Completing the **encryption/decryption task** was tedious.
   - Strongly agree
   - Agree
   - Neither agree nor disagree
   - Disagree
   - Strongly disagree
8. Completing the **encryption/decryption task** was challenging.

- Strongly agree
- Agree
- Neither agree nor disagree
- Disagree
- Strongly disagree

9. What parts of the **encryption/decryption task** were easy?
   - Text box
10. What parts of the **encryption/decryption task** were difficult?
    - Text box

A.1.2 Keygen task specific questions

1. Recall your experiences with the task where you were expected to expected to generate an encryption key and store it securely (**key generation and storage task**).
2. I completed the **key generation and storage task** correctly.
   - I am not confident.
   - I am slightly confident.
   - I am somewhat confident.
   - I am moderately confident.
   - I am absolutely confident.
3. I completed the **key generation and storage task** securely.
   - I am not confident.
   - I am slightly confident.
   - I am somewhat confident.
   - I am moderately confident.
   - I am absolutely confident.
4. The documentation was helpful in completing the **key generation and storage task**.
   - Strongly agree
   - Agree
   - Neither agree nor disagree
   - Disagree
   - Strongly disagree
5. Completing the **key generation and storage task** was frustrating.
   - Strongly agree
   - Agree
   - Neither agree nor disagree
   - Disagree
   - Strongly disagree
6. Completing the **key generation and storage task** was fun.
   - Strongly agree
   - Agree
   - Neither agree nor disagree
   - Disagree
   - Strongly disagree
7. Completing the **key generation and storage task** was tedious.
   - Strongly agree
   - Agree
   - Neither agree nor disagree
   - Disagree
   - Strongly disagree
8. Completing the **key generation and storage task** was challenging.
   - Strongly agree
   - Agree
   - Neither agree nor disagree
   - Disagree
   - Strongly disagree
9. What parts of the **key generation and storage task** were easy?
   - Text box
10. What parts of the **key generation and storage task** were difficult?
    - Text box

A.1.3 Study specific questions

1. Are you aware of a specific library or other resource you would have preferred to use to generate functional and secure code? If yes, please list them.
   - Yes [Text box]
   - No
2. Have you used or seen this assigned library before?
   - I have used the assigned library before. (e.g. worked on a project with assigned library)
   - I have seen code from the assigned library but not used it myself. (e.g. worked on a project with the library but someone else wrote the code)
   - I have neither used nor seen the assigned library before.
   - I don't know
3. Have you written or seen code for tasks similar to the assigned tasks before?
   - I have written similar code. (e.g. worked on a project that included a similar task)
   - I have seen similar code but have not written it myself. (e.g. worked on a project that included a similar task but someone else wrote the code)
   - I have never written nor seen code for similar tasks.
   - I don't know

A.1.4 System Usability Scale

1. We asked you to use an assigned library. To what extent do you agree with each of the following statements in reference to your assigned library and it's documentation: (Strongly agree, Agree, Neither agree nor disagree, Disagree, Strongly disagree)
   - I think that I would like to use this library frequently.

- I found this library unnecessarily complex.
- I thought this library was easy to use.
- I think that I would need the support of a technical person to be able to use this library.
- I found the various functions in this library were well integrated.
- I found this library fun to use. Regardless of what you felt please select strongly agree.
- I thought there was too much inconsistency in this library.
- I would imagine that most people would learn to use this library very quickly.
- I found this library very cumbersome to use.
- I felt very confident using this library.
- I needed to learn a lot of things before I could get going with this library.

A.1.5 Acar Usability Scale

1. We asked you to use an assigned library. To what extent do you agree with each of the following statements in reference to your assigned library and it's documentation: (Strongly agree, Agree, Neither agree nor disagree, Disagree, Strongly disagree)
    - I had to understand how most of the assigned library works in order to complete the tasks.
    - It would be easy and require only small changes to change parameters or configuration later without breaking my code.
    - After doing these tasks, I think I have a good understanding of the assigned library overall.
    - I only had to read a little of the documentation for the assigned library to understand the concepts that I needed for these tasks.
    - The names of classes and methods in the assigned library corresponded well to the functions they provided.
    - It was straightforward and easy to implement the given tasks using the assigned library.
    - When I accessed the assigned library documentation, it was easy to find useful help.
    - In the documentation, I found helpful explanations.
    - In the documentation, I found helpful code examples.
    - When I made a mistake, I got a meaningful error message/exception.
    - Using the information from the error message/exception, it was easy to fix my mistake.
    - Using the information from the error message/exception, it was hard to fix. Please select strongly disagree.

A.1.6 SSD-SES

1. During this portion of the survey, you will be shown hypothetical software development tasks. Please rate your level of confidence in completing the following software development tasks. (I am not confident, I am slightly confident, I am somewhat confident, I am moderately confident, I am absolutely confident, I do not understand the question)
    - I can perform a threat risk analysis (e.g. likelihood of vulnerability, impact of exploitation, etc.)
    - I can identify potential security threats to the system.
    - I can identify common attack techniques used by attackers.
    - I can identify potential attack vectors in the environment the system interacts with (e.g., hardware, libraries, etc.).
    - I can identify common vulnerabilities of a programming language.
    - I can design software to quarantine an attacker if a vulnerability is exploited.
    - I can mimic potential threats to the system.
    - I can evaluate security controls on the system's interfaces/interactions with other software systems.
    - I can evaluate security controls on the system's interfaces/interactions with other hardware systems.
    - I can communicate security assumptions and requirements to other developers on the team to ensure vulnerabilities are not introduced due to misunderstandings.
    - I can communicate system details with other developers to ensure a thorough security review of the code.
    - I can discuss lessons learned from internal and external security incidents to ensure all development team members are aware of potential threats.
    - I can effectively communicate identified security issues and the cost/risk trade-off associated with deciding whether or not to fix the problem to organization leadership.
    - I can communicate functionality needs to security experts to get recommendations for secure solutions (e.g., secure libraries, design patterns, and platforms).
    - I know the appropriate point of contact/response team in my organization to contact if a vulnerability in production code is identified.
    - I can perform security assessments. Regardless of your actual answer, please select I am absolutely confident.

A.1.7 General technical background

1. Including education, how long have you been programming? (In years)
   - Text box
2. Including education, how long have you been programming in Python? (In years)
   - Text box
3. Are you currently employed in a role that requires programming?
   - Yes
   - No
   - Maybe
4. (If yes or maybe to above) Is writing code in Python part of your primary job?
   - Yes
   - No
   - Maybe
5. (If yes or maybe to above) Not including education, how long have you been programming professionally? (In years)
   - Text box
6. (If yes or maybe to above) Not including education, how long have you been programming in Python professionally? (In years)
   - Text box
7. (If yes or maybe to above) Which of the following job roles describe you? (Please select all that apply)
   - Developer
   - Administrator
   - DevOps Engineer
   - Academic researcher/Scientist
   - Data science/Machine learning specialist
   - Educator
   - Engineer
   - Manager/Team lead
   - None
   - Other [Text box]
8. How did you learn to code? (Please select all that apply)
   - Self-taught
   - Online class
   - College/University
   - On-the-job training
   - Professional certification program
   - Coding bootcamp
   - I did not learn to code
   - Other [Text box]
9. How do you rate your knowledge of software security?
   - Very high
   - Above average
   - Average
   - Below average
   - Very low
10. Which of the following statements describe the secure programming training that you have received? (Please select all that apply)
    - I received secure programming training through an event organized by my employer
    - I learned secure programming concepts while working
    - I received secure programming training at school/-college/university
    - I received secure programming training at a workshop/seminar
    - I received secure programming training with online courses
    - I am self-taught
    - I have never received secure programming training
11. How many total years of experience do you have in computer security? (Experience includes years at work or studying in a security-related field)
    - Text box

A.1.8 Demographics

1. Please select the gender with which you most closely identify:
   - Man
   - Woman
   - Non-binary
   - Another gender/prefer to self-describe [Text box]
   - Prefer not to answer
2. What is your age in years?
   - Text box
3. Please specify your ethnicity. (Please select all that apply)
   - White
   - Hispanic or Latino
   - Black or African American
   - American Indian or Alaskan Native
   - Asian
   - Native Hawaiian or Pacific Islander
   - Prefer to self-describe [Text box]
   - Prefer not answer
4. Please select your highest completed education level.
   - Some high school
   - High school diploma/GED
   - Some college, no degree
   - Associate's degree
   - Bachelor's degree
   - Master's degree
   - Doctoral degree
   - Prefer not to answer
5. (If college or above) What was your primary field of study?
   - Computer science
   - IT security/Cyber security
   - Other engineering disciplines

- Never declared a major
- Other [Text box]

6. What is your country of residence?
   - Text box

A.2 Screening survey

A.2.1 General background

1. How long have you been programming?
   - Less than 1 year
   - 1 - 2 years
   - 2 - 5 years
   - More than 5 years
2. Are you currently a student?
   - Yes
   - No
3. (If yes to above) Are you currently majoring in something that requires programming?
   - Yes
   - Maybe
   - No
4. (If yes or maybe to above) What is your major?
   - Text box
5. Are you currently employed in a role that requires programming?
   - Yes
   - Maybe
   - No
6. (If yes or maybe to above) What is your occupation?
   - Text box
7. Please rate your proficiency with the following languages:
   - Java (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)
   - C (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)
   - C++ (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)
   - Python (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)
   - Rust (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)
   - Ruby (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)
   - Javascript (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)
   - OCaml (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)

A.2.2 Screener from Danilova et al. [15]

1. Which of the following websites do you most frequently use as an aid when programming?
   - Wikipedia
   - LinkedIn
   - StackOverflow
   - Memory Alpha
   - I have not used any of the websites above for programming.
   - I don't program
2. Choose the answer that best fits the description of a compiler's function.
   - Refactoring code
   - Connecting to the network
   - Aggregating user data
   - Translating code into executable instructions
   - Collecting user data
   - I don't know
3. Choose the answer that best fits the definition of a recursive function.
   - A function that runs for infinite time
   - A function that does not have a return value
   - A function that can be called from other functions
   - A function that calls itself
   - A function that does not require an input
   - A function that interprets cursive handwriting
   - I don't know
4. Which of these values could be assigned to a variable with the type Boolean?
   - Small
   - Solid
   - Quadratic
   - Red
   - True
   - I don't know
5. Answer the next two questions using the following snippet:

```python
def func(example):
    x = len(example)
    out = ""
    for i in range(x):
        out = out + example[x - i - 1]
    return out

print(func("hello world"))
```

6. Referring to the above code snippet, what is the parameter of the function?
   - out
   - example
   - for i in range(x)

- Outputting a string
- x = len(example)
- I don’t know

7. Please select the returned value of the pseudocode above:
    - hello world
    - hello world 10
    - dlrow olleh
    - world hello
    - HELLO WORLD
    - hello world hello world hello world hello world
    - I don’t know