Security Developer Studies with GitHub Users: Exploring a Convenience Sample

Yasemin Acar, *Leibniz University Hannover;* **Christian Stransky,** *CISPA, Saarland University;* **Dominik Wermke,** *Leibniz University Hannover;* **Michelle Mazurek,** *University of Maryland, College Park;* **Sascha Fahl,** *Leibniz University Hannover*

https://www.usenix.org/conference/soups2017/technical-sessions/presentation/acar

This paper is included in the Proceedings of the Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-39-3

Open access to the Proceedings of the Thirteenth Symposium on Usable Privacy and Security is sponsored by USENIX.

Security Developer Studies with GitHub Users: Exploring a Convenience Sample

Yasemin Acar,*† Christian Stransky,† Dominik Wermke,*† Michelle L. Mazurek,‡ and Sascha Fahl*†
*Leibniz University Hannover; †CISPA, Saarland University; ‡University of Maryland
{acar,wermke,fahl}@sec.uni-hannover.de; stransky@cs.uni-saarland.de; mmazurek@umd.edu

ABSTRACT

The usable security community is increasingly considering how to improve security decision-making not only for end users, but also for information technology professionals, including system administrators and software developers. Recruiting these professionals for user studies can prove challenging, as, relative to end users more generally, they are limited in numbers, geographically concentrated, and accustomed to higher compensation. One potential approach is to recruit active GitHub users, who are (in some ways) conveniently available for online studies. However, it is not well understood how GitHub users perform when working on security-related tasks. As a first step in addressing this question, we conducted an experiment in which we recruited 307 active GitHub users to each complete the same security-relevant programming tasks. We compared the results in terms of functional correctness as well as security, finding differences in performance for both security and functionality related to the participant's self-reported years of experience, but no statistically significant differences related to the participant's self-reported status as a student, status as a professional developer, or security background. These results provide initial evidence for how to think about validity when recruiting convenience samples as substitutes for professional developers in security developer studies.

1. INTRODUCTION

The usable security community is increasingly considering how to improve security decision-making not only for end users, but for information technology professionals, including system administrators and software developers [1, 2, 9, 10, 39]. By focusing on the needs and practices of these communities, we can develop guidelines and tools and even redesign ecosystems to promote secure outcomes in practice, even when administrators or developers are not security experts and must balance competing priorities.

One common approach in usable security and privacy research is to conduct an experiment, which can allow researchers to investigate causal relationships (e.g., [5,8,13,36]). Other non-field-study mechanisms, such as surveys and interview studies, are also common. For research concerned with the general population of end users, recruitment for these studies can be fairly straightforward, via online recruitment platforms such as Amazon Mechanical Turk or via local methods such as posting flyers and advertising on email lists or classified-ad services like Craigslist. These approaches generally yield acceptable sample sizes at an affordable cost.

Recruiting processes for security developer studies, however, are less well established. For in-lab studies, professional developers may be hard to contact (relative to the general public), may not be locally available outside of tech-hub regions, may have demanding schedules, or may be unwilling to participate when research compensation is considerably lower than their typical hourly rate. For these reasons, studies involving developers tend to have small samples and/or to rely heavily on university computer-science students [2, 3, 15, 34, 35, 39]. To our knowledge, very few researchers have attempted large-scale online security developer studies [1, 3].

To date, however, it is not well understood how these different recruitment approaches affect research outcomes in usable security and privacy studies. The empirical software engineering community has a long tradition of conducting experiments with students instead of professional developers [29] and has found that under certain circumstances, such as similar level of expertise in the task at hand, students can be acceptable substitutes [27]. These studies, however, do not consider a security and privacy context; we argue that this matters, because security and privacy tasks differ from general programming tasks in several potentially important ways. First, because security and privacy are generally secondary tasks, it can be dangerous to assume they exhibit similar characteristics as general programming tasks. For example, relative to many general programming tasks, it can be especially difficult for a developer to directly test that security is working. (For example, how does one observe that a message is correctly encrypted?) Second, a portion of professional developers are self-taught, so their exposure to security and privacy education may differ importantly from university students' [32].

The question of how to recruit for security studies of developers in order to maximize validity is complex but important. In this study, we take a first step toward answering it: We report on a experiment (n=307) comparing GitHub contributors completing the same security-relevant

Symposium on Usable Privacy and Security (SOUPS) 2017, July 12–14, 2017, Santa Clara, California.

tasks. For this experiment, we take as a case study the approach (which we used in prior work [1]) of recruiting active developers from GitHub for an online study. All participants completed three Python-programming tasks spanning four security-relevant concepts, which were manually scored for functionality and security. We found that participants across all programming experience levels were similarly inexperienced in security, and that professional developers reported more programming experience than university students. Being a professional did not increase a participant's likelihood of writing functional or secure code statistically significantly. Similarly, self-reported security background had no statistical effect on the results. Python experience was the only factor that significantly increased the likelihood of writing both functional and secure code. Further work is needed to understand how participants from GitHub compare to those recruited more traditionally (e.g., students recruited using flyers and campus e-mail lists, or developers recruited using meetup websites or researchers' corporate contacts). Nonetheless, our findings provide preliminary evidence that at least in this context, similarly experienced university students can be a valid option for studying professionals developers' security behaviors.

2. RELATED WORK

We discuss related work in two key areas: user studies with software developers and IT professionals focusing on security-relevant topics, and user studies with software developers and IT professionals that do not focus on security but do discuss the impact of participants' level of professionalism on the study's validity.

Studies with Security Focus. In [2] we present a laboratory study on the impact of information sources such as online blogs, search engines, official API documentation and StackOverflow on code security. We recruited both computer science students (40) and professional Android developers (14). We found that software development experience had no impact on code security, but previous participation in security classes had a significant impact. That study briefly compares students to professionals, finding that professionals were more likely to produce functional code but no more likely to produce secure code; however, that work does not deeply interrogate differences between the populations and the resulting implications for validity. In [1], we conducted an online experiment with GitHub users to compare the usability of cryptographic APIs; that work does not distinguish different groups of GitHub users.

Many studies with a security focus rely primarily on students. Yakdan et al. [39] conducted a user study to measure the quality of decompilers for malware analysis. Participants included 22 computer-science students who had completed an online bootcamp as well as 9 professional malware analysts. Scandariato et al. [28] conduct a controlled experiment with 9 graduate students, all of whom had taken a security class, to investigate whether static code analysis or penetration testing was more successful for finding security vulnerabilities in code. Layman et al. [22] conducted a controlled experiment with 18 computer-science students to explore what factors are used by developers to decide whether or not to address a fault when notified by an automated fault detection tool. Jain and Lindqvist [15] conducted a laboratory study with 25 computer-science students (5 graduate; 20 undergraduate) to investigate a new, more privacy-friendly location API for Android application developers and found that, when given the choice, developers prefer the more privacy-preserving API. Barik et al. [4] conducted an eye-tracking study with undergraduate and graduate university students to investigate whether developers read and understand compiler warning messages in integrated development environments.

Studies that use professional developers are frequently qualitative in nature, and as such can effectively make use of relatively small sample sizes. Johnson et al. [17] conducted interviews with 20 real developers to investigate why software developers do not use static analysis tools to find bugs in software, while Xie et al. [38] conducted 15 semi-structured interviews with professional software developers to understand their perceptions and behaviors related to software security. Thomas et al. [34] conducted a laboratory study with 28 computer-science students to investigate interactive code annotations for access control vulnerabilities. As follow up, Thomas et al. [35] conducted an interview and observation-based study with professional software developers using snowball sampling. They were able to recruit 13 participants, paying each a $25 gift card, to examine how well developers understand the researchers' static code analysis tool ASIDE. Johnson et al. [16] describe a qualitative study with 26 participants including undergraduate and graduate students as well as professional developers. Smith et al. [31] conducted an exploratory study with five students and five professional software developers to study the questions developers encounter when using static analysis tools.To investigate why developers make cryptography mistakes, Nadi et al. [25]surveyed 11 Stack Overflow posters who had asked relevant questions. A follow-up survey recruited 37 Java developers via snowball sampling, social media, and email addresses drawn from GitHub commits. This work does not address demographic differences, nor even whether participants were professional software developers, students, or something else.

A few online studies of developers have reached larger samples, but generally for short surveys rather than experimental tasks. Balebako et al. [3] studied the privacy and security behaviors of smartphone application developers; they conducted 13 interviews with application developers and an online survey with 228 application developers. They compensated the interviewees with $20 each, and the online survey participants with a $5 Amazon gift card. Witschey et al. [37] survey hundreds of developers from multiple companies (snowball sampling) and from mailing lists to learn their reasons for or against the use of security tools.

Overall, these studies suggest that reaching large numbers of professional developers can be challenging. As such, understanding the sample properties of participants who are more readily available (students, online samples, convenience samples) is an aspect of contextualizing the valuable results of these studies. In this paper, we take a first step in this direction by examining in detail an online sample from GitHub.

Studies without Security Focus. In the field of Empirical Software Engineering, the question whether or not students can be used as substitutes for developers when experimenting is of strong interest. Salman et al. [27] compared students and developers for several (non-security-related)

tasks, and found that the code they write can be compared if they are equally inexperienced in the subject they are working on. When professionals are more experienced than students, their code is better across several metrics. Hoest et al. [14] compare students and developers across assessment (not coding) tasks and find that under certain conditions, e.g., that students be in the final stretches of a Master's program, students can be used as substitutes for developers. Carver et al. [7] give instructions on how to design studies that use students as coding subjects. McMeekin et al. [23] find that different experience levels between students and professionals have a strong influence on their abilities to find flaws in code. Sjoeberg et al. [29] systematically analyze a decade's worth of studies performed in Empirical Software Engineering, finding that eighty-seven percent of all subjects were students and nine percent were professionals. They question the relevance for industry of results obtained in studies based exclusively on student recruits. Smith et al. [30] perform post-hoc analysis on previously conducted surveys with developers to identify several factors software researchers can use to increase participation rates in developer studies. Murphy-Hill et al. [24] enumerate dimensions which software engineering researchers can use to generalize their findings.

3. METHODS

We designed an online, between-subjects study to compare how effectively developers could quickly write correct, secure code using Python. We recruited participants, all with Python experience, who had published source code at GitHub.

Participants were assigned to complete a set of three short programming tasks using Python: an encryption task, a task to store login credentials in an SQLite database, and a task to write a routine for a URL shortener service. Each participant was assigned the tasks in a random order (no task depended on completing a prior task). We selected these tasks to provide a range of security-relevant operations while keeping participants' workloads manageable.

After finishing the tasks, participants completed an exit survey about the code they wrote during the study, as well as their educational background and programming experience. Two researchers coded participants' submitted code for functional correctness and security.

All study procedures were approved by the Ethics Review Board of Saarland University, the Institutional Review Board of the University of Maryland and the NIST Human Subjects Protection Office.

3.1 Language selection

We elected to use Python as the programming language for our experiment, as it is widely used across many communities and offers support for all kinds of security-related APIs, including cryptography. As a bonus, Python is easy to read and write, is widely used among both beginners and experienced programmers, and is regularly taught in universities. Python is the third most popular programming language on GitHub, trailing JavaScript and Java [12]. Therefore, we reasoned that we would be able to recruit sufficient professional Python developers and computer science students for our study.

3.2 Recruitment

As a first step to understanding security-study behavior of GitHub committers, we recruited broadly from GitHub, the popular source-code management service. To do this, we extracted all Python projects from the GitHub Archive database [11] between GitHub's launch in April 2008 and December 2016, yielding 798,839 projects in total. We randomly sampled 100,000 of these repositories and cloned them. Using this random sample, we extracted email addresses of 80,000 randomly chosen Python committers. These committers served as a source pool for our recruitment.

We emailed these GitHub users in batches, asking them to participate in a study exploring how developers use Python. We did not mention security or privacy in the recruitment message. We mentioned that we would not be able to compensate them, but the email offered a link to learn more about the study and a link to remove the email address from any further communication about our research. Each contacted GitHub users was assigned a unique pseudonymous identifier (ID) to allow us to correlate their study participation to their GitHub statistics separately from their email address.

Recipients who clicked the link to participate in the study were directed to a landing page containing a consent form. After affirming that they were over 18, consented to the study, and were comfortable with participating in the study in English, they were introduced to the study framing. We did not restrict participation to those with security expertise because we were interested in the behavior of non-security-experts encountering security as a portion of their task.

To explore the characteristics of this sample, the exit questionnaire included questions about whether they were currently enrolled in an undergraduate or graduate university program and whether they were working in a job that mainly involved Python programming. We also asked about years of experience writing Python code, as well as whether the participant had a background in computer security.

3.3 Experimental infrastructure

For this study, we used an experimental infrastructure we developed, which is described in detail in our previous work [1, 33].

We designed the experimental infrastructure with certain important features in mind:

- A controlled study environment that would be the same across all participants, including having pre-installed all needed libraries.
- The ability to capture all code typed by our participants, capture all program runs and attendant error messages, measure time spent working on tasks, and recognize whether or not code was copied and pasted.
- Allowing participants to skip tasks and continue on to the remaining tasks, while providing information on why they decided to skip the task.

To achieve these goals, the infrastructure uses Jupyter Notebooks (version 4.2.1) [19], which allow our participants to write, run, and debug their code in the browser, without having to download or upload anything. The code runs on our

server, using our standardized Python environment (Python 2.7.11). This setup also allows us to frequently snapshot participants' progress and capture copy-paste events. To prevent interference between participants, each participant was assigned to a separate virtual machine running on Amazon's EC2 service. Figure 1 shows an example Notebook.

We pre-installed many popular Python libraries for accessing an SQLite database, dealing with string manipulation, storing user credentials, and cryptography. Table 9 in Appendix C lists all libraries we provided. We tried to include as many relevant libraries as possible, so that every participant could work on the tasks using their favorite libraries.

The tasks were shown one at a time, with a progress indicator showing how many tasks remained. For each task, participants were given buttons to "Run and test" their code, and to move on using "Solved, next task" or "Not solved, but next task." (A "Get unstuck" button was also provided in case the participant accidentally sent Python into an infinite loop or otherwise crashed the Python interpreter running in the Notebook.) After completing (or skipping) all tasks, the participant was redirected to the exit survey.

3.4 Exit survey

Once all tasks had been completed or abandoned, the participants were directed to a short exit survey (cf. Appendix A). We asked for their opinions about the tasks they had completed: Did they think they had solved them? How did they perceive the tasks' difficulty? Did they think their solution was secure? We also were interested in whether they thought about security or privacy when working on the tasks. Finally, we wanted to know whether our participants had worked on similar programming problems in the past. For these task-specific questions, we used our infrastructure to display the participant's code for the corresponding task for their reference. We also asked several questions about demographic information and programming experience, to allow us to distinguish categories of participants.

3.5 Task design

We designed tasks that were short enough so that the uncompensated participants would be likely to complete them before losing interest, but still complex enough to be interesting and allow for some mistakes. Most importantly, we designed the tasks to model real-world security and privacy problems that Python developers could reasonably be expected to encounter. While these tasks of course do not represent all possible security tasks, we think they provide an interesting variety for analysis.

URL Shortener

We asked our participants to write code for a method that could be part of a URL shortening service such as bit.ly [1] or Google's URL shortener goo.gl. [2]

URL Shortener Task

Description:
You are asked to develop code for a URL shortening service similar to `https://goo.gl/`. Users of this service will provide you URLs such as `https://en.wikipedia.org/wiki/History_of_the_Internet`. As a result your service should return a shortened URL such as `http://foo.bar/1xf2`. In this task we would like you to implement a method `shortenURL` that is called for every input URL. The output of this method is a shortened URL for the input URL.

When is the problem solved?
A shortened URL is returned. Please print the output of the method to the console.

While this task does not directly have security implications, we were mainly interested in whether, e.g., they used well established approaches such as message digest or random number generators to generate a short URL, or whether they invented their own algorithm.

Credential Storage

We asked our participants to store login credentials, namely a username and password, in a database backend.

Credential Storage Task

Description:
You are asked to develop a web-application backend that stores login credentials (i.e., usernames and passwords) for the web application's users. In this task, we would like you to implement a method `storeCredentials` that is called for every user at account registration. New login credentials are appended to a local SQLite database. Assume that the username and password are given as HTTP POST parameters to your method. Although we are not asking you to implement the `verifyCredentials` method for authenticating users at this time, assume that you will also be writing that method, so you can choose the storage format within the database. We have prepared a SQLite database named "db.sqlite" containing a table "users" and five text columns, "column1", "column2", "column3", "column4", "column5". You can use any or all of these columns as needed to store users' login credentials; you do not have to use all columns to solve the task.

When is the problem solved?
The credentials are stored in the database file.

This task has direct security implications: we were mainly interested in whether participants followed security best practices. Best practices to store user credentials in a database include hashing and salting the password instead of storing it in plaintext and using some sort of input sanitization (e.g. parameterized instead of raw SQL queries) for SQL queries to prevent SQL injection attacks.

[1] cf. `https://bitly.com/`
[2] cf. `https://goo.gl/`

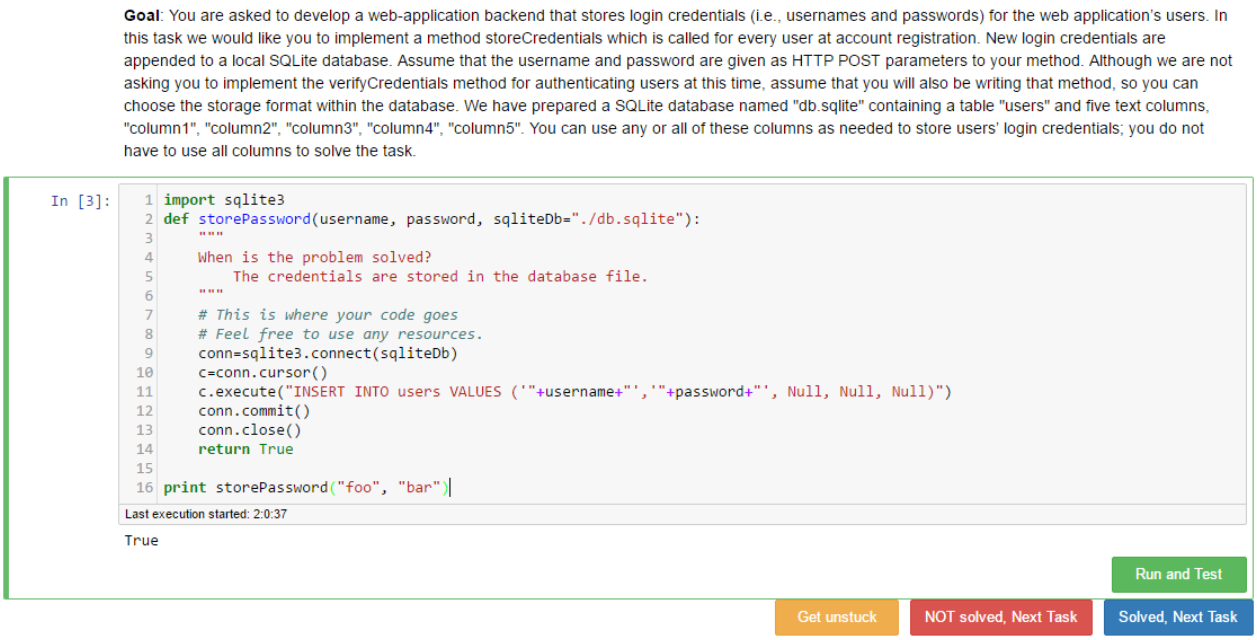

Figure 1: An example of the study's task interface.

String Encryption

We asked participants to write code to encrypt and decrypt a string.

String Encryption Task

Description:
You are asked to write code that is able to encrypt and decrypt a string.

When is the problem solved?
The input string is encrypted and decrypted afterwards. You should see the encrypted and decrypted string in the console.

In this task we were mainly interested in whether participants wrote secure cryptographic code, e.g., choosing secure algorithms, strong key sizes, and secure modes of operation.

For each task, we provided stub code and some comments with instructions about how to work on the task. The code stubs were intended to make the programming task as clear as possible and to ensure that we would later easily be able to run automated unit tests to examine functionality. The code stubs also helped to orient participants to the tasks.

We told participants that "you are welcome to use any resources you normally would" (such as documentation or programming websites) to work on the tasks. We asked participants to note any such resources as comments to the code, for our reference, prompting them to do so when we detected that they had pasted text and/or code into the Notebook.

3.6 Evaluating participant solutions

We used the code submitted by our participants for each task, henceforth called a *solution*, as the basis for our analysis. We evaluated each participant's solution to each task for both functional correctness and security. Every task was independently reviewed by two coders, using a content analysis approach [21] with a codebook based on our knowledge of the tasks and best practices. Differences between the two coders were resolved by discussion. We briefly describe the codebook below.

Functionality. For each programming task, we assigned a participant a functionality score of 1 if the code ran without errors, passed the unit tests and completed the assigned task, or 0 if not.

Security. We assigned security scores only to those solutions which were graded as functional. To determine a security score, we considered several different security parameters. A participant's solution was marked secure (1) only if their solution was acceptable for every parameter; an error in any parameter resulted in a security score of 0.

URL Shortener

For the URL shortening task, we checked how participants generated a short URL for a given long URL. We were mainly interested in whether participants relied on well-established mechanisms such as message digest algorithms (e.g. the SHA1 or SHA2 family) or random number generators, or if they implemented their own algorithms. The idea behind this evaluation criterion is the general recommendation to rely on well-established solutions instead of reinventing the wheel. While adhering to this best practice is advisable in software development in general, it is particularly crucial for writing security- or privacy-relevant code (e.g., use established implementations of cryptographic algorithms instead of re-implementing them from scratch). We also considered the reversibility of the short URL as a security parameter (reversible was considered insecure). We did not incorporate whether solutions were likely to produce collisions (i. e. produce the same short URL for different in-

put URLs) or the space of the URL-shortening algorithm (i. e. how many long URLs the solution could deal with) as security parameters: we felt that given the limited time frame, asking for an optimal solution here was asking too much.

Credential Storage

For the credential storage task, we split the security score in two. One score (password storage) considered how participants stored users' passwords. Here, we were mainly interested whether our participants followed security best practices for storing passwords. Hence, we scored the plain text storage of a password as insecure. Additionally, applying a simple hash algorithm such as MD5, SHA1 or SHA2 was considered insecure, since those solutions are vulnerable to rainbow table attacks. Secure solutions were expected to use a salt in combination with a hash function; however, the salt needed to be random (but not necessarily secret) for each password to withstand rainbow table attacks. Therefore, using the same salt for every password was considered insecure. We also considered the correct use of HMACs [20] and PBKDF [18] as secure.

The second security score (SQL input) considered how participants interacted with the SQLite database we provided. For this evaluation, we were mainly interested whether the code was vulnerable to SQL injection attacks. We scored code that used raw SQL queries without further input sanitization as insecure, while we considered using prepared statements secure.[3]

String Encryption

For string encryption, we checked the selected algorithm, key size and proper source of randomness for the key material, initialization vector and, if applicable, mode of operation. For symmetric encryption, we considered ARC2, ARC4, Blowfish, (3)DES and XOR as insecure and AES as secure. We considered ECB as an insecure mode of operation and scored Cipher Block Chaining (CBC), Counter Mode (CTR) and Cipher Feedback (CFB) as secure. For symmetric key size, we considered 128 and 256 bits as secure, while 64 or 32 bits were considered insecure. Static, zero or empty initialization vectors were considered insecure. For asymmetric encryption, we considered the use of OAEP/PKCS1 for padding as secure. For asymmetric encryption using RSA, we scored keys larger than or equal to 2048 bits as secure.

3.7 Limitations

As with any user study, our results should be interpreted within the context of our limitations.

Choosing an online rather than an in-person laboratory study allowed us less control over the study environment and the participants' behavior. However, it allowed us to recruit a diverse set of developers we would not have been able to obtain for an in-person study.

Recruiting using conventional recruitment strategies, such as posts at university campuses, on Craigslist, in software development forums or in particular companies would likely have limited the number and variety of our participants. As a result, we limited ourselves to active GitHub users. We believe that this resulted in a reasonably diverse sample, but of course GitHub users are not necessarily representative of developers more broadly, and in particular students and professionals who are active on GitHub may not be representative of students and professionals overall. The small response rate compared to the large number of developers invited also suggests a strong opt-in bias. Comparing the set of invited GitHub users to the valid participants suggests that more active GitHub users were more likely to participate, potentially widening this gap. As a result, our results may not generalize beyond the GitHub sample. However, all the above limitations apply equally across different properties of our participants, suggesting that comparisons between the groups are valid.

Because we could not rely on a general recruitment service such as Amazon's Mechanical Turk, managing online payment to developers would have been very challenging; further, we would not have been able to pay at an hourly rate commensurate with typical developer salaries. As a result, we did not offer our participants compensation, instead asking them to generously donate their time for our research.

We took great care to email each potential participant only once, to provide an option for an email address to opt out of receiving any future communication from us, and to respond promptly to comments, questions, or complaints from potential participants. Nonetheless, we did receive a small number of complaints from people who were upset about receiving unsolicited email.[4]

Some participants may not provide full effort or many answer haphazardly; this is a particular risk of all online studies. Because we did not offer any compensation, we expect that few participants would be motivated to attempt to "cheat" the study rather than simply dropping out if they were uninterested or did not have time to participate fully. We screened all results and attempted to remove any obviously low-quality results (e.g., those where the participant wrote negative comments in lieu of real code) before analysis, but cannot discriminate with perfect accuracy. Further, our infrastructure based on Jupyter Notebooks allowed us to control, to an extent, the environment used by participants; however, some participants might have performed better had we allowed them to use the tools and environments they typically prefer. However, these limitations are also expected to apply across all participants.

4. STUDY RESULTS

We were primarily interested in comparing the performances of different categories of participants in terms of functional and secure solutions. Overall, we found that students and professionals report differences in experience (as might be expected), but we did not find significant differences between them in terms of solving our tasks functionally or securely.

4.1 Statistical Testing

In the following subsections, we analyze our results using regression models as well as non-parametric statistical testing. For non-regression tests, we primarily use the Mann-Whitney-U test (MWU) to compare two groups with nu-

[3]While participants could have manually sanitized their SQL queries, we did not find a single solution that did that.

[4]Overall, we received 13 complaints.

meric outcomes, and X^2 tests of independence to compare categorical outcomes. When expected values per field are too small, we use Fisher's exact test instead of X^2.

Here, we explain the regression models in more detail. The results we are interested in have binary outcomes; therefore, we use *logistic* regression models to analyze those results. the consideration whether an insecure task counts as *dangerous*, i.e. whether it is functional, insecure and the programmer thinks it is secure, is also binary and therefore analyzed analogously. As we consider results on a per-task basis, we use a mixed model with a random intercept; this accounts for multiple measures per participant. For the regression analyses, we select among a set of candidate models with respect to the Akaike Information Criterion (AIC) [6]. All candidate models include which task is being considered, as well as the random intercept, along with combinations of optional factors including years of Python experience, student and professional status, whether or not the participant reported having a security background, and interaction effects among these various factors. These factors are summarized in Table 1. For all regressions, we selected as final the model with the lowest AIC.

The regression outcomes are reported in tables; each row measures change in the dependent variable (functionality, security, or security perception) related to changing from the *baseline* value for a given factor to a different value for the same factor (e.g., changing from the encryption task to the URL shortening task). The regressions output odds ratios (O.R.) that report on change in likelihood of the targeted outcome. By construction, O.R.=1 for baseline values. For example, Table 2 indicates that the URL shortening task was 0.45× as likely to be functional as the baseline string encryption task. In each row, we also report a 95% confidence interval (C.I.) and a p-value; statistical significance is assumed for p≤.05, which we indicate with an asterisk (*). For both regressions, we set the encryption task to be the baseline, as it was used similarly in previous work [1].

4.2 Participants

We sent 23,661 email invitations in total. Of these, 3,890 (16.4%) bounced and another 447 (1.9%) invitees requested to be removed from our list, a request we honored. 16 invitees tried to reach the study but failed due to technical problems in our infrastructure, either because of a large-scale Amazon outage[5] during collection or because our AWS pool was exhausted during times of high demand.

A total of 825 people agreed to our consent form; 93 (11.3%) dropped out without taking any action, we assume because the study seemed too time-consuming. The remaining 732 participants clicked on the begin button after a short introduction; of these, 440 (60.1%) completed at least one task and 360 of those (81.8%) proceeded to the exit survey. A total of 315 participants completed all programming tasks and the exit survey. We excluded eight for providing obviously invalid results. From now on, unless otherwise specified, we report results for the remaining 307 valid participants, who completed all tasks and the exit survey.

[5]Some participants were affected by this Amazon EC2 outage: `https://www.recode.net/2017/3/2/14792636/amazon-aws-internet-outage-cause-human-error-incorrect-command`.

We classified these 307 participants into students and professionals according to their self-reported data. If a participant reported that they work at a job that mainly requires writing code, we classified them as a professional. If a participant reported being an undergraduate or graduate student, we classified them as a student. It was possible to be classified as either only a professional, only a student, both, or neither. The 307 valid participants includes 254 total professionals, 25 undergraduates, and 49 graduate students. 53 participants were both students and professionals; 32 participants were neither students nor developers. Due to the small sample size, we treated undergraduates and graduate students as one group for further analysis.

The 307 valid participants reported ages between 18 and 81 years (mean: 31.6; sd: 7.7) [Student: 19-37, mean: 25.3, sd: 5.2 - Professional: 18-54, mean: 32.9, sd: 6.7], and most of them reported being male (296 - Student: 21 - Professional 194). All but one of our participants (306) had been programming in general for more than two years and 277 (Student: 18, Professional: 186) had been programming in Python for more than two years. The majority (288 - Student: 20, Professional: 188) said they had no IT-security background nor had taken any security classes.

We compared students to non-students and professionals to non-professionals for security background and years of Python experience. (We compared them separately because some participants are both students and professionals, or are neither.) In both cases, there was no difference in security background (due to small cell counts, we used Fisher's exact test; both with $p \approx 1$). Professionals had significantly more experience in Python than non-professionals, with a median 7 years of experience compared to 5 (MWU, $W = 5040$, $p = 0.004$). Students reported significantly less experience than non-students, with median 5 years compared to 7 years (MWU, $W = 10963$, $p < 0.001$).

The people we invited represent a random sample of GitHub users — however, our participants are a small, self-selected subset of those. We were able to retrieve metadata for 192 participants; for the remainder, GitHub returned a 404 error, which most likely means that the account was deleted or set to private after the commit we crawled was pushed to GitHub. We compare these 192 participants to the 12117 invited participants for whom we were able to obtain GitHub metadata.

Figure 2 illustrates GitHub statistics for all groups (for more detail, see Table 8 in the Appendix). Our participants are slightly more active than the average GitHub user: They have a median of 3 public gists compared to 2 for invited GitHub committers (MWU, W = 1045300, p = 0.01305); they have a median of 28 public repositories compared to 21 for invited participants (MWU, W = 1001200, p < 0.001); they all follow a median of 3 committers (MWU, W = 1142100, p = 0.66); and they are followed by a similar number of committers (10 for participants, 11 for invited; MWU, W = 1146100, p = 0.73).

4.3 Functionality

We evaluated the functionality of the code our participants wrote while working on the programming tasks. Figure 3 illustrates the distribution of functionally correct solutions between tasks and across professional developers and uni-

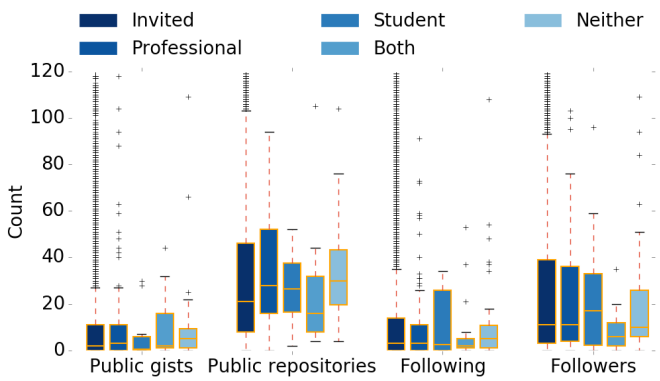

Figure 2: Boxplots comparing our invited participants (a random sample from GitHub) with those who provided valid participation. The center line indicates the median; the boxes indicate the first and third quartiles. The whiskers extend to ±1.5 times the interquartile range. Outliers greater than 150 were truncated for space.

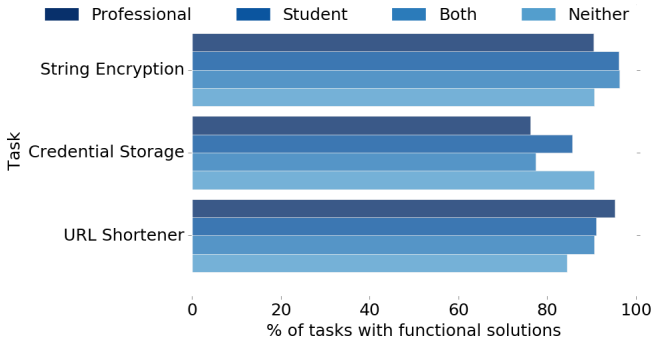

Figure 3: Functionality results per task, split by students vs. professional developers.

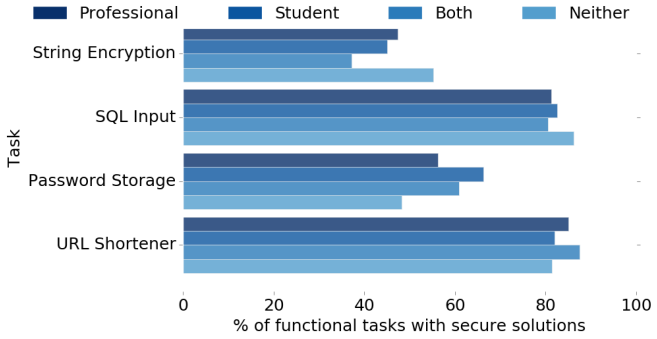

Figure 4: Security results per task, split by students vs. professional developers.

versity students. Overall, professionals got 720 of 804 tasks correct (89.6%), while students got 71 of 84 correct (84.5%); participants who were both students and professionals got 181 of 212 (85.4%) correct, while participants who were neither succeeded in 114 of 128 (89.1%) cases.

Table 2 shows the results of the regression model for functionality. The final model does not include developer or student status, security background, or any interaction effects, suggesting that these factors are not important predictors of functional success. Python experience, on the other hand, did produce a statistically significant effect: each additional year of experience corresponds to on average a 10% increase in likelihood of a correct solution. Comparing tasks, the password storage task proved most difficult: participants were only 0.45× as likely to complete it as to complete the baseline string encryption task. Results for the URL shortening task were comparable to the baseline.

To assess the fit of our regression model, we use Nagelkerke's method [26] to compute a pseudo-R^2 value, somewhat analogous to the standard coefficient of determination commonly used with ordinary linear regression. We find that, relative to a null model that includes only the random (per-participant) effect, our model produces a pseudo-R^2 of 0.07; this is not a particularly strong fit, reflecting the fact that there are potentially many unmeasured covariates, such as the specifics of a participant's prior programming experience and education.

4.4 Security

We evaluated the security of the code based on the codebook described in Section 3.6. In this section, we talk about four tasks instead of three, as the credentials storage task had two security relevant components that we account for individually: secure password digest and SQL input validation (see Section 3.6 for details).

Figure 4 illustrates the distribution of secure solutions between tasks and across professional developers vs. university students. Altogether, professionals got 493 of 720 tasks correct (68.5%), while students got 48 of 71 correct (67.6%); participants who were both students and professionals got 119 of 181 (65.7%) correct, while participants who were neither succeeded in 77 of 114 (67.5%) cases.

Table 3 lists the results of the final security regression model. This model had Nagelkerke pseudo-R^2 of 0.183, which is a fairly strong fit for an uncontrolled experiment with potential unmeasured factors.

As with the functionality results, none of developer status, student status, security background, nor any interactions, appear in the final model. This again suggests that these factors do not meaningfully predict security success. As before, more Python experience is associated with more success: this time, each year of additional experience adds about 5% to the likelihood of a secure solution. Comparing tasks, string encryption proved significantly more difficult to complete securely than any other task. Password storage was associated with about 2× higher likelihood of success. Both these tasks were significantly harder than SQL input validation and URL shortening. (The non-overlapping confidence intervals indicate significant difference from password storage as well as from the baseline string encryption task). SQL input validation and URL shortening were each about 8× easier to secure than string encryption.

4.4.1 *Security Perception*

We asked participants, for each task, whether they believed their result was secure. In this section, we analyze the incidence of what we call *dangerous* solutions: solutions that

Factor	Description	Baseline
Required		
Task	The performed tasks	String encryption
Participant	Random effect accounting for repeated measures	n/a
Optional		
Python experience	Python programming experience in years, self-reported.	n/a
Security background	True or false, self-reported.	False
Developer	True or false, self-reported.	False
Student	True or false, self-reported.	False
Python experience × task		False:String encryption
Python experience × developer		False:False
Python experience × student		False:False
Developer × task		False:String encryption
Student × task		False:String encryption

Table 1: Factors used in regression models. Categorical factors are individually compared to the baseline. Final models were selected by minimum AIC; candidates were defined using all possible combinations of optional factors, with the required factors included in every candidate.

Factor	O.R.	C.I.	p-value
URL shortener	0.45	[0.22, 0.89]	0.022*
Credentials storage	0.22	[0.11, 0.42]	<0.001*
Python experience	1.10	[1.02, 1.19]	0.014*

Table 2: Results of the final logistic regression model examining functionality of tasks for participants. Odds ratios (O.R.) indicate relative likelihood of succeeding. Statistically significant factors indicated with *. See Table 1 for further details.

Factor	O.R.	C.I.	p-value
URL shortener	8.03	[5.14, 12.53]	<0.001*
Password storage	2.34	[1.6, 3.43]	<0.001*
SQL input	7.69	[4.89, 12.09]	<0.001*
Python experience	1.05	[1.01, 1.1]	0.020*

Table 3: Results of the final logistic regression model examining security of tasks for participants. Odds ratios (O.R.) indicate relative likelihood of succeeding. Statistically significant factors indicated with *. See Table 1 for further details.

are functionally correct and where the participant believes the result is secure, but our analysis indicates that it is not. In a sense, this represents a worst-case scenario, where a developer may confidently release insecure code unwittingly.

Table 4 details how perceptions of security connect to evaluated security. Across tasks, 154 of 1228 (12.5%) solutions were classified as dangerous; happily, dangerous solutions were least common of the four classes, but this rate is still higher than we might hope.

Table 5 reports on a regression model with whether or not a solution is classified as dangerous as the binary outcome. The final model contains no optional factors at all. This indicates that none of Python experience, security background, professional status, or student status is a good predictor of a dangerous outcome. Indeed, the Nagelkerke pseudo-R^2 for this model is only 0.049, which reflects that we did not measure important additional factors.

Our regression model suggests that string encryption, which was most difficult to secure, was (unsurprisingly) also associated with significantly higher likelihood of dangerous solutions than the SQL input and URL shortening tasks. Encryption, however, was comparable to password digests, which also have a cryptographic component. In a prior experiment, we found that about 20% of cryptographic tasks fell into this dangerous category [2].

4.4.2 Investigating Security Errors

We also examined patterns in the types of security errors made by our participants across tasks. Note that these patterns reflect only functional but insecure solutions. In all cases, the same solution may have more than one security error, so percentages generally total to more than 100%.

URL Shortening

First, we consider the URL shortening task. The most common security error (11 cases, 23.0%) was participants who implemented their URL shortening feature using an algorithm that allows an attacker to easily predict the long URL for a given short URL. An example is the use of Base 64 to derive a "short" URL from a given long URL. Although we did not consider keyspace as a security parameter, we briefly review the keyspace generated by participants with functional solutions to this task. 104 participants (37.4%) selected a shortening approach with an unlimited keyspace. The remaining 174 solutions had an average keyspace of 74.1 bits (median 48, standard deviation 6.1). The average for professionals (82.0 bits, median 48) was higher than for students (62.5 bits, median 48), participants who were both students and professionals (58.5 bits, median 36) and participants who were neither (60.6 bits, median 36).

Password Storage

Next, we consider insecure password storage. Here the most common error was hashing the password without using a proper salt, leaving the stored password vulnerable to rainbow-table attacks (74 cases, 77.1%). The second most common error was storing the plain password (45 cases, 46.9%). A total of 19 (19.8%) participants used a static salt

Category	Encryption	Password Storage	URL shortener	SQL input	Total
Dangerous (Perception Secure & Scoring Insecure)	41 (13.4%)	57 (18.6%)	17 (5.5%)	39 (12.7%)	154
Harmless Misperception (Perception Insecure & Scoring Secure)	49 (16.0%)	31 (10.1%)	156 (50.8%)	64 (20.8%)	300
True Positives (Perception Secure & Scoring Secure)	82 (26.7%)	131 (42.7%)	75 (24.4%)	149 (48.5%)	437
True Negatives (Perception Insecure & Scoring Insecure)	135 (44.0%)	88 (28.7%)	59 (19.2%)	55 (17.9%)	337

Table 4: Detailed distribution of perceived and actual security within functional solutions, broken out per task. Percentages are as a function of each task; for example, 13.4% of all encryption solutions were categorized as dangerous.

Factor	O.R.	C.I.	p.value
URL shortener	0.25	[0.12, 0.52]	<0.001*
Password storage	1.16	[0.7, 1.93]	0.565
SQL input	0.53	[0.29, 0.97]	0.038*

Table 5: Results of the final logistic regression model examining perceived security and actual security. Odds ratios (O.R.) indicate relative likelihood of being insecure. Statistically significant factors indicated with *. See Table 1 for further details.

instead of a random salt. Seven (7.3%) participants used MD5, while six (6.3%) used SHA-1 family hashes. Instead of using a one way hash function, four (4.2%) used encryption to secure the password. This is highly discouraged, since an attacker who can gain access to the decryption key is able to recover plain text passwords. These results are detailed in Table 6.

SQL Query

For the SQL query task, 44 (97.8%) of the participants used raw SQL queries instead of prepared statements, leaving their implementation vulnerable to SQL injection attacks. Interestingly, no participant tried to implement their own SQL query sanitization solution.

Encryption

For the string encryption task, one important decision participants made was the choice of cryptographic library (cf. Table 9 for the libraries that came pre-installed). 118 (40.4% of all functional solutions) of the participants used a cryptographic library that was designed with usability in mind, which reduces the necessity to select (and potential make an error with) parameters like algorithm, mode of operation, key size, initialization vector, and padding scheme (cryptography.io: 103, PyNacl: 15, PySodium: 1). 93 participants (31.8% of all functional solutions) chose a more conventional library (PyCrypto: 93), and 73 (25.0% of all functional solutions) used no third-party library at all.

Overall, 15 (12.7%) of participants who applied a usable library made a security error, while 49 (52.7%) of the participants who used a conventional library made a security error. All participants but one who used usable libraries used secure algorithms, modes of operation, and key sizes; the other 14 who made an error used a static initialization vector. Users of conventional cryptographic libraries mostly used a static initialization vector (31 cases, 63.3% of error cases), used an insecure mode of operation (11, 22.4% of error cases), or chose an insecure algorithm (7, 14.3% of error cases). These results indicate that usable libraries do reduce errors, and they are in line with the errors we identified in a prior experiment [1]. These results are detailed in Table 7.

```
1  def encrypt(plainText):
2      return ''.join([chr(ord(c) + n % 5) for
       n, c in enumerate(plainText)])
3
4  def decrypt(cipherText):
5      return ''.join([chr(ord(c) - n % 5) for
       n, c in enumerate(cipherText)])
6
7  stringToEncrypt = "ThisIsAnExample"
8  encryptedString = encrypt(stringToEncrypt)
9  print encryptedString
10 decryptedString = decrypt(encryptedString)
11 print decryptedString
```

Listing 1: Substitution cipher solution as written by a professional developer participant.

Among participants who did not apply cryptography effectively, 20 used Base64 to encode their plaintext instead of encrypting it, and 43 implemented a very basic substitution cipher like Rot13. An example is shown in Listing 1.

5. DISCUSSION AND CONCLUSIONS

In our online quasi-experiment with 307 GitHub participants, we measured functionality and security outcomes across Python programming tasks. We came into the experiment hypothesizing that whether or not a participant wrote code professionally or as a student would impact at least the functional correctness of their code. However, we found that neither student nor professional status (self-reported) was a significant factor for functionality, security, or security perception. We were also surprised to learn that self-reported security background was equally unimportant. (Note that only small numbers of participants reported that they were exclusively students or that they had a security background, which may affect these results).

We did, however, find a significant effect for Python experience: Each year of experience corresponded to 10% more likelihood of getting a functional result and a 5% better chance of getting a secure result. Differences in experience across students and professionals were significant: Students reported a median of 5 years of experience, compared to 7 for professionals. (On the other hand, experience did not

	Plain password	MD5 hash	SHA1 hash	No salt	Static salt	Raw SQL	Not stored
Professionals	24 (14.0%)	3 (1.7%)	4 (2.3%)	40 (23.3%)	15 (8.7%)	29 (16.9%)	1 (0.6%)
Student	4 (25.0%)	0 (0.0%)	0 (0.0%)	6 (37.5%)	1 (6.2%)	3 (18.8%)	0 (0.0%)
Both	8 (19.5%)	2 (4.9%)	2 (4.9%)	14 (34.1%)	2 (4.9%)	8 (19.5%)	0 (0.0%)
Neither	9 (31.0%)	2 (6.9%)	0 (0.0%)	14 (48.3%)	1 (3.4%)	4 (13.8%)	0 (0.0%)
Total	45 (17.4%)	7 (2.7%)	6 (2.3%)	74 (28.7%)	19 (7.4%)	44 (17.1%)	1 (0.4%)

Table 6: Types of security errors found in functional solutions (and their percentages) by professional, student, both or neither for the password storage task. See Subsection 3.5 for task details and Subsection 3.6 for codebook details.

Library	Used	Weak Algo	Weak Mode	Static IV
Professionals				
No library	44 (22.8%)	42 (21.8%)	0 (0.0%)	0 (0.0%)
cryptography.io	71 (36.8%)	0 (0.0%)	0 (0.0%)	10 (5.2%)
pyCrypto	65 (33.7%)	5 (2.6%)	9 (4.7%)	23 (11.9%)
PyNaCl	10 (5.2%)	0 (0.0%)	0 (0.0%)	1 (0.5%)
Other	3 (1.6%)	3 (1.6%)	0 (0.0%)	0 (0.0%)
Student				
No library	8 (42.1%)	8 (42.1%)	0 (0.0%)	0 (0.0%)
cryptography.io	5 (26.3%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
pyCrypto	6 (31.6%)	1 (5.3%)	0 (0.0%)	1 (5.3%)
Both				
No library	17 (33.3%)	17 (33.3%)	0 (0.0%)	0 (0.0%)
cryptography.io	16 (31.4%)	0 (0.0%)	0 (0.0%)	3 (5.9%)
pyCrypto	15 (29.4%)	1 (2.0%)	2 (3.9%)	5 (9.8%)
PyNaCl	1 (2.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
pySodium	1 (2.0%)	1 (2.0%)	0 (0.0%)	0 (0.0%)
Other	1 (2.0%)	1 (2.0%)	0 (0.0%)	0 (0.0%)
Neither				
No library	6 (20.7%)	6 (20.7%)	0 (0.0%)	0 (0.0%)
cryptography.io	11 (37.9%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
pyCrypto	7 (24.1%)	0 (0.0%)	0 (0.0%)	2 (6.9%)
PyNaCl	4 (13.8%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
Other	1 (3.4%)	1 (3.4%)	0 (0.0%)	0 (0.0%)
Total				
No library	75 (25.7%)	73 (25.0%)	0 (0.0%)	0 (0.0%)
cryptography.io	103 (35.3%)	0 (0.0%)	0 (0.0%)	13 (4.5%)
pyCrypto	93 (31.8%)	7 (2.4%)	11 (3.8%)	31 (10.6%)
PyNaCl	15 (5.1%)	0 (0.0%)	0 (0.0%)	1 (0.3%)
pySodium	1 (0.3%)	1 (0.3%)	0 (0.0%)	0 (0.0%)
Other	5 (1.7%)	5 (1.7%)	0 (0.0%)	0 (0.0%)

Table 7: Types of security errors found in functional solutions (and their percentages) by professional, student, both or neither for the string encryption task. Participant categories are subdivided by the cryptographic library they opted to use. See Subsection 3.5 for task details and Subsection 3.6 for codebook details.

appear to matter for security perception.) This accords well with previous results within the empirical software engineering community (cf. Section 2), which suggest that student and professional developer participants' expertise should be similar to produce similar results. While expertise with Python in our study differs significantly between students and professional developers, their security and privacy expertise is similar (in both cases quite low). At least within GitHub then, it seems that students and professionals can be equally useful for studying usable security and privacy problems, particularly if overall experience is controlled for.

In addition to the small sample size, we speculate that the very similar results across students and professional developers can be accounted for in part because writing security-related code is not something the average software developer deals with on a regular basis, nor is security education a strong focus at many universities teaching computer science. We hypothesize, therefore, that overall these results — experience matters somewhat, but professional status on its own does not — would continue to hold for student and professional populations recruited more traditionally, at local universities and via professional networks. We suspect, however, that typically local university students may have less experience than students recruited from GitHub. Further research is needed to validate these hypotheses.

We found the recruitment strategy of emailing GitHub developers to be convenient in many ways: We were able to recruit many experienced professionals quickly and at a low cost. In addition, many participants expressed to us how much they enjoyed the challenge of our tasks and the opportunity to contribute to our research. However, it does have important drawbacks: we received complaints about unsolicited email from 13 invited GitHub committers and were generally subject to a small opt-in rate. We also found that our participants were slightly more active and therefore not quite representative of the GitHub population; represen-

tativeness for professionals (or students) in general is considerably less likely. Overall, the practice of sending unsolicited emails was not ideal, and is unlikely to be sustainable over many future studies. Instead, we plan in the future to develop a GitHub application that would allow developers who are interested in contributing to research to opt in to study recruitment requests, which would benefit both these developers and the research community.

6. ACKNOWLEDGEMENTS

The authors would like to thank Mary Theofanos and the anonymous reviewers for providing feedback; Rob Reeder for shepherding the paper and guiding us in a substantial change of direction; Andrea Dragan and Anne Andrews for help managing multi-institution ethics approvals; Simson Garfinkel and Doowon Kim for contributing to the study infrastructure; and all of our participants for their contributions. This work was supported in part by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA), and by the U.S. Department of Commerce, National Institute for Standards and Technology, under Cooperative Agreement 70NANB15H330.

7. REFERENCES

[1] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. Comparing the Usability of Cryptographic APIs. In *Proc. 38th IEEE Symposium on Security and Privacy (SP'17)*. IEEE, 2017.

[2] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. You Get Where You're Looking For: The Impact of Information Sources on Code Security. In *Proc. 37th IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 2016.

[3] R. Balebako, A. Marsh, J. Lin, and J. Hong. The Privacy and Security Behaviors of Smartphone App Developers. In *Proc. Workshop on Usable Security (USEC'14)*. The Internet Society, 2014.

[4] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin. Do Developers Read Compiler Error Messages? In *Proc. 39th IEEE International Conference on Software Engineering (ICSE'17)*. IEEE, 2017.

[5] C. Bravo-Lillo, S. Komanduri, L. F. Cranor, R. W. Reeder, M. Sleeper, J. Downs, and S. Schechter. Your Attention Please: Designing Security-decision UIs to Make Genuine Risks Harder to Ignore. In *Proc. 9th Symposium on Usable Privacy and Security (SOUPS'13)*. USENIX Association, 2013.

[6] K. P. Burnham. Multimodel Inference: Understanding AIC and BIC in Model Selection. *Sociological Methods & Research*, 33(2):261–304, 2004.

[7] J. Carver, L. Jaccheri, S. Morasca, and F. Shull. Issues in using students in empirical studies in software engineering education. In *Proc. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (Healthcom'03)*. IEEE, 2003.

[8] S. Fahl, M. Harbach, T. Muders, M. Smith, and U. Sander. Helping Johnny 2.0 to encrypt his Facebook conversations. In *Proc. 8th Symposium on Usable Privacy and Security (SOUPS'12)*. USENIX Association, 2012.

[9] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith. Rethinking SSL Development in an Appified World. In *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)*. ACM, 2013.

[10] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *Proc. 38th IEEE Symposium on Security and Privacy (SP'17)*. IEEE, 2017.

[11] GitHub Archive, Nov. 2 2016. visited.

[12] GitHut: A Small Place to discover languages in github, Nov. 2 2016. visited.

[13] M. Harbach, M. Hettig, S. Weber, and M. Smith. Using Personal Examples to Improve Risk Communication for Security and Privacy Decisions. In *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'14)*. ACM, 2014.

[14] M. Höst, B. Regnell, and C. Wohlin. Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering*, 5(3):201–214, 2000.

[15] S. Jain and J. Lindqvist. Should I Protect You? Understanding Developers' Behavior to Privacy-Preserving APIs. In *Proc. Workshop on Usable Security (USEC'14)*. The Internet Society, 2014.

[16] B. Johnson, R. Pandita, J. Smith, D. Ford, S. Elder, E. Murphy-Hill, S. Heckman, and C. Sadowski. A Cross-Tool Communication Study on Program Analysis Tool Notifications. In *Proc. 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. ACM, 2016.

[17] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proc. 35th IEEE International Conference on Software Engineering (ICSE'13)*. IEEE, 2013.

[18] S. Josefsson. PKCS #5: Password-Based Key Derivation Function 2 (PBKDF2) Test Vectors, Jan. 2011.

[19] Jupyter notebook, Nov. 2 2016. visited.

[20] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication, Feb. 1997.

[21] K. Krippendorff. *Content Analysis: An Introduction to Its Methodology (2nd ed.)*. SAGE Publications, 2004.

[22] L. Layman, L. Williams, and R. S. Amant. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *Proc. First International Symposium on Empirical Software Engineering and Measurement (ESEM'07)*. IEEE, 2007.

[23] D. A. McMeekin, B. R. von Konsky, M. Robey, and D. J. Cooper. The significance of participant experience when evaluating software inspection techniques. In *Proc. 20th Australian Conference on Software Engineering (ASWEC'09)*. IEEE, 2009.

[24] E. Murphy-Hill, D. Y. Lee, G. C. Murphy, and J. McGrenere. How Do Users Discover New Tools in Software Development and Beyond? *Computer Supported Cooperative Work (CSCW)*, 24(5):389–422,

2015.

[25] S. Nadi, S. Krüger, M. Mezini, and E. Bodden. "Jumping Through Hoops": Why do Java Developers Struggle With Cryptography APIs? In *Proc. 37th IEEE International Conference on Software Engineering (ICSE'15)*. IEEE, 2016.

[26] N. J. Nagelkerke. A note on a general definition of the coefficient of determination. *Biometrika*, 78(3):691–692, 1991.

[27] I. Salman, A. T. Misirli, and N. Juristo. Are students representatives of professionals in software engineering experiments? In *Proc. 37th IEEE International Conference on Software Engineering (ICSE'15)*. IEEE Press, 2015.

[28] R. Scandariato, J. Walden, and W. Joosen. Static analysis versus penetration testing: A controlled experiment. In *Proc. 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013.

[29] D. I. K. Sjoeberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N. K. Liborg, and A. C. Rekdal. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, 31(9):733–753, 2005.

[30] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird, and T. Zimmermann. Improving developer participation rates in surveys. In *Proc. 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE'13)*. IEEE, 2013.

[31] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *Proc. 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015.

[32] Stack overflow - developer survey results, June 10 2017. visited.

[33] C. Stransky, Y. Acar, D. C. Nguyen, D. Wermke, E. M. Redmiles, D. Kim, M. Backes, S. Garfinkel, M. L. Mazurek, and S. Fahl. Lessons Learned from Using an Online Platform to Conduct Large-Scale, Online Controlled Security Experiments with Software Developers. In *Proc. 10th USENIX Workshop on Cyber Security Experimentation and Test (CSET'17)*. USENIX Association, 2017.

[34] T. Thomas, B. Chu, H. Lipford, J. Smith, and E. Murphy-Hill. A study of interactive code annotation for access control vulnerabilities. In *Proc. 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'15)*. IEEE, 2015.

[35] T. W. Thomas, H. Lipford, B. Chu, J. Smith, and E. Murphy-Hill. What Questions Remain? An Examination of How Developers Understand an Interactive Static Analysis Tool. In *Proc. 2nd Workshop on Security Information Workers (WSIW'16)*. USENIX Association, 2016.

[36] B. Ur, P. G. Kelley, S. Komanduri, J. Lee, M. Maass, M. L. Mazurek, T. Passaro, R. Shay, T. Vidas, L. Bauer, N. Christin, and L. F. Cranor. How does your password measure up? The effect of strength meters on password creation. In *Proc. 21st Usenix Security Symposium (SEC'12)*. USENIX Association, 2012.

[37] J. Witschey, O. Zielinska, A. Welk, E. Murphy-Hill, C. Mayhorn, and T. Zimmermann. Quantifying developers' adoption of security tools. In *Proc. 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015.

[38] J. Xie, H. R. Lipford, and B. Chu. Why do programmers make security errors? In *Proc. 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'11)*. IEEE, 2011.

[39] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In *Proc. 37th IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 2016.

APPENDIX

A. EXIT SURVEY QUESTIONS

Task-specific questions: Each task has these questions

On a five-point scale, how much do you agree with the following statements: [strongly agree, agree, neither agree nor disagree, disagree, strongly disagree]

- The task was difficult. (for each task)
- I am confident my solution is correct. (for each task)
- I am confident my solution is secure. (for each task)

What makes this solution either secure or insecure? (free text per task)

When you performed the task, were you thinking about security or privacy? (for each task)

- yes, a lot
- yes, a little
- no

What specifically? (For each task) [free text]

Have you written similar code or come across similar problems in the past? (For each task).

- yes
- sort of
- no

Tell us about it. When was it and what did you do; did you do something differently? [free text]

Demographics and past experience

Check all that apply: Have you ever taken a computer security class?

- at an undergraduate level
- at a graduate level
- via online learning
- via professional training

- another way [specify]
- no, but I took a class that had security as one major component or module
- no

How many computer security classes total have you taken? [input a number]

When did you last take a computer security class? [input a year]

Check all that apply: Do you have experience working in computer security or privacy outside of school?

- Professionally (you got paid to do it)
- As a hobby
- No
- Other [specify]

Check all that apply: Have you ever taken a Python programming class?

- at an undergraduate level
- at a graduate level
- via online learning
- via professional training
- another way [specify]
- no, but I took a class that had Python as one major component or module
- no

How many total Python classes have you taken? [input a number]

When did you last take a Python class [input a year]

Do you have experience programming in Python outside of school?

- Professionally (you got paid to do it)
- As a hobby
- No
- Other [specify]

For how many years have you been programming in Python? [number]

How many Python projects have you worked on in the past? [number]

When did you last work on a Python project? [year]

For how many years have you been programming in general (not just in Python)? [number]

How did you primarily learn to program? (Choose one)

- Self-taught
- In a university / as part of a degree
- In an online learning program
- In a professional certification program
- On the job
- Other [specify]

What is your gender?

- Male
- Female
- Other
- Prefer not to answer

What is your age? [number]

Are you currently a student?

- Undergraduate
- Graduate
- Professional certification program
- Other [specify]
- Not a student

Are you currently employed at a job where programming is a critical part of your job responsibility? [yes/no]

What country did you (primarily) grow up in? [list of countries]

What is your native language (mother tongue)? [list of languages]

B. GITHUB DEMOGRAPHICS

Table 8 compares demographics for invited users vs. participants.

C. INSTALLED PYTHON LIBRARIES

Table 9 lists the Python libraries we pre-installed in the study infrastructure.

	Invited	Valid - Pros	Valid - Students	Valid - Both	Valid - Neither
Hireable	20.5%	19.4%	40.0%	30.6%	23.5%
Company listed	39.4%	43.4%	30.0%	38.9%	17.6%
URL to blog	48.0%	47.3%	40.0%	63.9%	58.8%
Biography added	14.1%	21.7%	20.0%	16.7%	29.4%
Location provided	62.0%	69.8%	50.0%	69.4%	29.4%
GitHub profile creation (days ago, median)	2158	2148	1712	2101	2191
GitHub profile last update (days ago, median)	22	20	23	18	14
Minimal/Maximal age	—	18 / 54	19 / 37	19 / 43	24 / 81
Average age (Std)	—	32.9 (6.7)	25.3 (5.2)	27.5 (4.7)	35.2 (12.7)
More than 2 years programming experience	—	99.5%	100.0%	100.0%	100.0%
More than 2 years Python experience	—	92.5%	85.7%	81.2%	88.7%
Security background	—	6.5%	4.8%	5.7%	6.2%
Male/Female[1]	—	96.5% / 1.5%	100.0% / 0.0%	94.3% / 5.7%	96.9% / 0.0%

[1] the remainder either answered "other" or prefer not to disclose their gender.

Table 8: GitHub demographics for invited users vs. our valid participants.

Library	Version
apsw	3.8.11.1.post1
backports-abc	0.5
backports.shutil-get-terminal-size	1.0.0
bcrypt	2.0.0
blinker	1.3
certifi	2016.9.26
cffi	1.9.1
chardet	2.3.0
configparser	3.5.0
cryptography	1.2.3
cryptography-vectors	1.2.3
decorator	4.0.10
ecdsa	0.13
entrypoints	0.2.2
enum34	1.1.6
file-encryptor	0.2.9
Flask	0.10.1
flufl.password	1.3
functools32	3.2.3.post2
idna	2.0
ipaddress	1.0.16
ipykernel	4.5.2
ipython	5.1.0
ipython-genutils	0.1.0
ipywidgets	5.2.2
itsdangerous	0.24
Jinja2	2.8
jsonschema	2.5.1
jupyter	1.0.0
jupyter-client	4.4.0
jupyter-console	5.0.0
jupyter-core	4.2.0
M2Crypto	0.22.6rc4
m2ext	0.1
macaron	0.3.1
MarkupSafe	0.23
mistune	0.7.3
nbconvert	4.2.0
nbformat	4.1.0

Library	Version
ndg-httpsclient	0.4.0
notebook	4.2.3
passlib	1.6.5
pathlib2	2.1.0
pexpect	4.2.1
pickleshare	0.7.4
prompt-toolkit	1.0.9
ptyprocess	0.5.1
pyasn1	0.1.9
pycparser	2.17
pycrypto	2.6.1
pycryptopp	0.6.0.12...
Pygments	2.1.3
pyinotify	0.9.6
PyNaCl	1.0.1
pyOpenSSL	0.15.1
pysodium	0.6.9.1
pysqlite	2.7.0
python-geohash	0.8.3
python-keyczar	0.715
python-mhash	1.4
pyzmq	16.0.2
qtconsole	4.2.1
requests	2.9.1
simplegeneric	0.8.1
singledispatch	3.4.0.3
six	1.10.0
smbpasswd	1.0.1
ssdeep	3.1
terminado	0.6
tlsh	0.2.0
tornado	4.4.2
traitlets	4.3.1
typing	3.5.3.0
urllib3	1.13.1
wcwidth	0.1.7
Werkzeug	0.10.4
widgetsnbextension	1.2.6
withsqlite	0.1

Table 9: Pre-installed libraries.