Deception Task Design in Developer Password Studies: Exploring a Student Sample

Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith,
University of Bonn, Germany

https://www.usenix.org/conference/soups2018/presentation/naiakshina

This paper is included in the Proceedings of the Fourteenth Symposium on Usable Privacy and Security.

August 12–14, 2018 • Baltimore, MD, USA

ISBN 978-1-931971-45-4

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

Deception Task Design in Developer Password Studies: Exploring a Student Sample

Alena Naiakshina
University of Bonn
naiakshi@cs.uni-bonn.de

Anastasia Danilova
University of Bonn
danilova@cs.uni-bonn.de

Christian Tiefenau
University of Bonn
tiefenau@cs.uni-bonn.de

Matthew Smith
University of Bonn
smith@cs.uni-bonn.de

ABSTRACT

Studying developer behavior is a hot topic for usable security researchers. While the usable security community has ample experience and best-practice knowledge concerning the design of end-user studies, such knowledge is still lacking for developer studies. We know from end-user studies that task design and framing can have significant effects on the outcome of the study. To offer initial insights into these effects for developer research, we extended our previous password storage study [42]. We did so to examine the effects of deception studies with regard to developers. Our results show that there is a huge effect - only 2 out of the 20 non-primed participants even attempted a secure solution, as compared to the 14 out of 20 for the primed participants. In this paper, we will discuss the duration of the task and contrast qualitative vs. quantitative research methods for future developer studies. In addition to these methodological contributions, we also provide further insights into why developers store passwords insecurely.

1. INTRODUCTION

Applying the philosophy and methods of usable security and privacy research to developers [31] is still a fairly new field of research. As such, the community does not yet have the body of experience concerning study design that it does for end-user studies. Many factors need to be considered when designing experiments. In what setting should they be conducted: a laboratory, online, or in the field? Who should the participants be: computer science students, or professional administrators and developers? Is a longitudinal study needed, or is a first contact study sufficient? Should a qualitative or quantitative approach be taken? How many participants are needed and can realistically be recruited? Is deception necessary to elicit unbiased behavior? How big do tasks need to be? And so forth. All these factors have an influence on the ecological validity of studies with developers. Thus, research is needed to analyze the effects of these design variables.

In this paper, we present a study exploring two of these design choices. First, we examine the effect of deception/priming on computer science students in a developer study.

USENIX Symposium on Usable Privacy and Security (SOUPS) 2018.
August 12–14, 2018, Baltimore, MD, USA.

To do so, we extended a developer study on password storage (primary study) using different study designs (meta-study) to evaluate the effects of the design.

In end-user studies, deception is a divisive topic. For instance, Haque et al. [32] argue that deception is necessary for password studies: "We did not want to give the participants any clue about our experimental motive because we expected the participants to spontaneously construct new passwords, exactly in the same way as they do in real life." However, Forget et al. [28] explicitly told their participants that they were studying passwords and asked participants to create them as they would in real life, in the hope of getting more realistic passwords. In an experiment to determine whether stating that the study is about passwords has an effect (i.e., priming the participants), Fahl et al. [20] found that there was no significant effect in an end-user study. Thus, there is evidence that deception is not needed for end-user studies. This is particularly relevant in terms of ethical considerations, since deception studies should only be used if absolutely necessary and the potential harm to participants must be weighed carefully.

We face similar questions when designing developer studies. For example, should we inform participants that we are studying the security of their password storage code and thus prime them, or do we need to use deception to gain insights into their "natural" behavior?

Second, we share our insights on the differences between our quantitative study and a qualitative exploration of password storage. One of the big challenges of developer studies is recruiting enough participants to conduct quantitative research. To examine this, we extended our qualitative password storage study [42] to implement a quantitative analysis and contrast the insights gained with both methods.

The rest of the paper is structured as follows. In section 2, we discuss related work. In section 3, we introduce our study methodology and explain how the study was extended. Section 4 discusses the limitations of our study and section 5 the ethical considerations. Section 6 contains the main hypotheses of our study. Section 7 presents the results and section 8 discusses the methodological contributions. Finally, section 9 summarizes the take-aways and section 10 concludes the paper.

2. RELATED WORK

This paper contributes to two distinct areas of research. The main contribution concerns the effect of priming/deception in usable security studies for developers. The related work on this topic is discussed in section 2.1. We also extend the body of knowledge on

developer studies specifically concerning password storage. Here, the related work is divided into multiple sections. Section 2.2 discusses other developer studies in general, section 2.3 focuses on developer studies concerning passwords, section 2.4 is about technical studies of password storage, and finally, section 2.5 discusses work on the usability of application program interfaces (APIs).

2.1 Priming in study design

In their research on website authentication protection mechanisms, Schechter et al. [47] explored the practice of deception in the form of a *priming effect*. They conducted a study with three groups. The participants of the first two groups were asked to role-play working with authentication data in a bank setting. One of these groups was thereby primed by receiving security-focused instructions. Fahl et al. [20] conducted a between-groups study with two variables (lab vs. online study, priming vs. non-priming). They specifically compared real-world password choices with passwords chosen by end-users in a study environment, considering priming and non-priming conditions. While the primed group was asked to behave as in real life when creating and managing passwords, the term "password" was not mentioned at all in the introductory text for the non-primed group. Neither Schechter et al. nor Fahl et al. found a significant effect for either the priming or non-priming conditions. However, both studies were conducted with end-users. Past research has shown that experts such as developers differ from end-users with regard to their mental models, behavior etc. [50, 34, 8]. Research on significant effects for developers concerning the priming and non-priming conditions does not yet exist.

In [42], we conducted a qualitative study (from which the present study acquired part of its data) with 20 computer science students, in order to investigate why developers fail with regard to password storage. Participants were asked to implement a registration task for a web application in 8 hours. We explored four scenarios: (1) priming (telling participants to consider password storage security) vs. (2) non-priming (deceiving participants by telling them the study was about API usability); and (3) web application framework with password storage support vs. (4) web application framework without password storage support. Our results indicated that frameworks offering only opt-in support for password storage and participants having strong background knowledge in software security practices were not sufficient for the production of secure software. Developers need to be told about when and how to use such security mechanisms. While the study in [42] was of a qualitative nature, we aimed at a more extensive study and invited 20 more participants for a quantitative analysis.

With consideration for the results and findings of previous studies, we offer initial insights into how developer security studies should be designed. Furthermore, we compare qualitative vs. quantitative studies and analyses, discuss time conditions for studies and examine participants' search behavior when working on a security-related task.

2.2 Developer studies

Acar et al. [9] conducted an online study with Python developers, recruited from GitHub (www.github.com). They were asked to use one of five cryptographic libraries to implement a set of security-related tasks. The main finding of this study was that simple APIs help developers to produce secure code; however, good documentation with a wide range of examples is still essential. For most of the tested libraries, security success was under 80%. Furthermore, 20% of the functional solutions were incorrectly rated by participants as being secure.

Acar et al. [10] conducted a between-subjects study to examine the impact of different documentation resources on the security of code. Fifty-four developers were given a skeleton Android app, which they had to extend in four security-related tasks. For assistance, they had access either to (1) Stack Overflow (www.stackoverflow.com), (2) books, (3) the official Android documentation, or (4) could freely choose which source to use. Programmers assigned to Stack Overflow produced less secure code. Furthermore, Fischer et al. [26] analyzed 1.3 million Android apps and found that 15.4% of them contained Stack Overflow source code. Of the analyzed source code, 97.9% contained at least one insecure code part.

Fahl et al. [23] interviewed software developers who implemented vulnerable applications regarding Secure Sockets Layer (SSL) issues. As a result, a framework was designed that prevented developers from producing insecure software in terms of SSL.

Stylos and Myers [48] investigated the relationship between secure code and the method placement of crypto APIs. They created two different versions of three APIs and asked programmers to solve three small tasks. For the same task, developers tended to use the same starting class. This resulted in faster task solution when using APIs with starting classes that referenced the class they needed.

Prechelt [44] investigated whether diverse programming languages (Java EE, Perl, PHP) or differences between the programmer teams are reflected in the security of the resulting code. For each programming language, they asked three programmer teams, comprising three professional developers each, to implement a web application in 30 h. The outcome was analyzed in terms of usability, functionality, reliability, structure, and security. They found the smallest within-platform variations for PHP.

2.3 Passwords - Developer studies

As it is often difficult to recruit professional developers for studies, Acar et al. [11] wanted to find out whether active GitHub users could be of interest for usable security studies. They conducted an online experiment with 307 GitHub users, who had to implement security-related tasks. One of these tasks considered credential storage. Neither the self-reported status as student or professional developer nor the participants' security background correlated with the functionality or security of their solutions. However, they found a significant effect for Python experience on functionality and security of program code.

Bau et al. [13] examined the vulnerability rate of web applications and programming language as well as developers in terms of career (start-up, freelancer) and their security background knowledge. For the start-up group, existing programs were analyzed. For the freelance group, eight compensated developers were invited to participate in a developer study. As compared to the start-up group, the freelancers were primed for security in the task description. With regard to secure password storage, it was found that there is a huge gap between the freelancers' knowledge and their actual implementation. Furthermore, the use of PHP and freelancers increased the software vulnerability rate.

Nadi et al. [41] studied the kinds of problems developers struggle with when using APIs. They analyzed the top 100 cryptographic questions on Stack Overflow as well as 100 randomly selected GitHub repositories that used Java crypto APIs. Within the analyzed projects, they found passwords being encrypted. This is a discouraged practice, which should be replaced by hashing. Afterwards, they conducted a study with 11 developers and a survey with 48 developers. Code templates, tools to catch common mistakes and

better documentation that includes examples were suggested for solving problems.

2.4 Passwords - Technical analysis

Bonneau and Preibusch [15] analyzed 150 websites and found they all lacked secure implementation choices. They did not use encryption, stored end-user passwords in plain text, or offered only little or no protection against brute-force attacks. This was particularly true for websites with few security incentives, such as newspapers.

Finifter and Wagner [24] analyzed nine implementations of the same web application, written in three different programming languages (Java, Perl, and PHP) in order to find correlations between the number of vulnerabilities and the programming language as well as the framework support for various aspects of security. They found no correlation between the security of web applications and the programming language. Automatic features offered by frameworks were an effective way of preventing vulnerabilities in general; however, this did not apply for secure password storage.

Egele et al. [19] analyzed more than 11 000 Android apps, with a focus on previously formulated security rules as well as password storage security. According to their findings, 88% violated at least one of those rules.

2.5 Usability of crypto APIs

While APIs are crucial for implementing secure applications that handle sensitive data, many of them seem to be too complex. As a conclusion to various examples [19, 40, 41], Green and Smith [31] presented ten principles for crypto APIs in order to reduce developer errors. Further, Gorski et al. [29] evaluated studies concerning API usability. They proposed eleven usability characteristics they consider necessary for secure APIs.

Lazar et al. [40] studied cryptographic vulnerabilities that were reported in the Common Vulnerabilities and Exposures (CVE) database. Of these, 83% were found to be a consequence of API misuse. They stated that no existing technique could prevent certain classes of mistakes.

3. METHODOLOGY

The aim of our study was to gain insight into the design of developer studies. To that end, we used two different kinds of independent variables (IVs). The first was on the meta-level, i.e., variables concerning study design. In our case, we had two meta-IVs: task design (priming and deception) and type of study (qualitative and quantitative). We refer to these as meta-variables of the meta-study. We also have an independent variable concerning the actual study subject, in our case the framework used to store passwords (JavaServer Faces [JSF] or Spring). We refer to this variable as the primary variable of the primary study.

The study presented in this paper is an extension of our previous qualitative study on password storage [42]. This quantitative study was planned at the same time to facilitate the analysis of the study type meta-variable, comparing the qualitative and quantitative approaches.

To summarize the qualitative study: participants were told that they should implement the user registration functionality for a social networking platform. Half the participants were instructed to use the Spring framework, which has built-in features for secure password storage. The other half was instructed to use JSF, a framework with manual support for password storage. This part of the design addressed the primary study. Additionally, half the participants were told the purpose of the study was to examine their password behavior and that they should store the passwords securely. The other half received a deceptive study description, which stated that the study was about the usability of APIs. For more detailed information and the exact phrasing of the tasks, see [42]. After the task was completed, a questionnaire was administered and semi-structured interviews were conducted. For the task description and the interviews, participants could choose their preferred language, either English or German. The survey, however, was in English and had to be answered in English. The study was set up for 8 h.

The main difference between the two studies was that in the qualitative study, the exit interviews were used to gain qualitative insights into the development process, while in the quantitative study, we used the survey responses and data gathered by the platform to conduct statistical testing. The hypotheses for this paper were developed before the qualitative analysis in [42] was started. This approach allowed us to gain insights into how a qualitative approach compares to a more quantitative approach.

In the combined study, we examined the following independent variables: for the primary study, the IV was the *framework* used for development (either Spring or JSF). For the meta-study, we used the IVs *priming* (deception or true purpose) and the *type of study* (qualitative or quantitative).

Participants for both studies were recruited together via a prescreening survey advertised through the computer science email list of the University of Bonn and flyers on the computer science campus. In total, 82 computer science students completed the questionnaire. Of these, 67 were invited to take part in the study. Seven of these were used for pilot studies, leaving 60 invited participants.

The first 20 participants were used for the qualitative study published in [42]. The remaining participants were used to extend the participant pool for this study. Although we had not planned to do a qualitative analysis on the remaining candidates, we conducted the exit interviews with all participants. This was done to treat all participants equally and to enable extending the qualitative analysis beyond the initially planned 20 in the event we did not reach saturation.

We removed two participants from the dataset of [42], JN1 and SP2. Due to a technical fault, the code history was not stored for JN1, and SP2 misunderstood the task so completely that no useful data was collected. This was not a big problem for the qualitative analysis but would have made the quantitative comparisons more complicated. Two random participants with a similar skill profile were selected as replacements. Of the remaining 30 invited participants, only 22 showed up. This left us with a total of 40 participants. Participation was compensated with 100 euros. Table 1 shows the demographics of all 40 participants. In the rest of the paper, we will present the quantitative analysis based on these 40 participants. The four conditions being tested are shown in section 3.1. In addition, we will contrast the qualitative findings in [42] with the quantitative findings.

3.1 Conditions

We conducted an experiment with 40 computer science students in order to explore whether task framing and different levels of framework support for password storage affect the security of software. Participants were asked to implement a registration process for a web application in a social network context, as described in [42]. The experiment was conducted under the following four conditions:

1. **Priming** - Participants were explicitly told to store the user

Gender	Male: 77.5%	Female: 15%	Prefer not to say: 7.5%
Ages	mean = 24.89	median = 25	sd = 2.89
Level of education	Bachelor: 30%	Master: 65%	Other: 5%
Study program	Computer Science: 82.5%	Media Informatics: 15%	Other: 2.5%
Country of Origin	Germany: 32.5% Iran: 5% Indonesia: 2.5% Finnland: 2.5%	India: 27.5% United States: 2.5% Turkey: 2.5% Uzbekistan: 2.5%	Syria: 5% Korea: 2.5% Pakistan: 2.5% Prefer not to say: 2.5%
Java experience	< 1 year : 42.5% 6-10 years: 5%	1-2 years: 27.5%	3-5 years: 25%

Table 1: Demographics of 40 participants.

passwords securely in the *Introductory Text* and in the *Task Description*.

2. **Non-priming** - Participants were told the study is about API usability, but were not explicitly asked for *secure* password storage.

3. **Framework with opt-in support for password storage** - Participants were advised to use a framework offering a secure implementation option, which could be used if they thought of it or found it. Spring was chosen as a representative framework [42].

4. **Framework with manual support for password storage** - Participants were advised to use a framework with the weakest level of support for password storage. Thus, they had to write their own salting and hashing code using just crypto primitives. JSF was considered as a suitable web framework in this case [42].

Java was selected as the programming language because it is one of the most popular and widely used programming languages for applications and web development [1, 7, 5, 3, 4, 6]; in addition, it is regularly taught at our university. Therefore, we reasoned that we would be able to recruit a sufficient sample of computer science students for our study.

Since a related study [11] has shown that self-reported skills of developers affect the study results, we used randomized condition assignments and counterbalanced for participants' skills reported in the pre-questionnaire (this is known as Randomized Block Design [37]). The pre-questionnaire can be found in Appendix A.

3.2 Deception

We examined the effect that concealing the true purpose of the study had as opposed to openly making it about secure password storage. Kimmel indicated three stages in which deception can be integrated: *subject recruitment, research procedure and post-research/application* [36, p.65]. In our study, we investigated whether participants made sure to store end-user passwords securely, if they were not explicitly told to do so in either the *Introductory Text* or the *Task Description* (non-primed group). In the recruiting phase, all candidates (primed and non-primed) were told the purpose of the study is API usability research (*"The goal of the study is to test the usability of different Java web development APIs."*). Consequently, we used deception in the *subject recruitment* and *research procedure* stages.

3.3 Experimental environment

The experiment was performed in an in-person laboratory, which allowed us to control the study environment and the participants' behavior. We created an instrumented Ubuntu distribution designed for developer studies that included code-specific tracking features. Thus, we were able to collect all data produced by the participants within the 8 h sessions (e.g., the web history and program code history). Every code snippet that was compiled was secured in a history folder. In addition to a video recording of the participants' desktops, the setup also allowed us to take frequent snapshots of their progress. In order to capture copy/paste events, we used *Glipper* [2], a clipboard manager for GNOME, which we modified slightly to meet our requirements (e.g., adding a time stamp to the events in a log file). In this manner, the study environment allowed us to identify all participants who copied and pasted code for password storage and the websites from which they received the code.

3.4 Survey

Before working on the task, participants filled out a short entry survey regarding their expectations for task difficulty. They also completed a self-assessment of their programming skills (see Appendix B). After finishing the implementation task, participants were required to complete an exit survey (see Appendix C). We asked participants for their demographics, security background knowledge, programming experience, and experience with the task and APIs. Furthermore, we asked open questions that could be answered with free text. Two coders independently coded the participants' answers by using Grounded-Theory and compared their final code books using the inter-coder agreement. The Cohen's kappa coefficient (κ) [18] for all themes was 0.78. A value above 0.75 is considered a good level of coding agreement [27].

To analyze the usability of the APIs, we applied the 11-question scale suggested by Acar et al. [9] (Appendix C.1), since it is more developer-oriented than the standard System Usability Scale (SUS) [16], which is more end-user oriented. Acar et al.'s usability scale is a combination of the cognitive dimensions framework [17], usability suggestions from Nielsen [43], and developer-related recommendations from Green and Smith [31].

3.5 Scoring code security

We used the same scoring system as was used in [42]. For each solution, we examined its **functionality and security**. We rated participants' solutions as functional if "an end user was able to register the Web application, meaning that his/her data provided through the interface was stored to a database" [42].

We used two measures to record the security of a participant's

solution. Every solution was rated on a scale from 0 to 7, according to the security score introduced in [42] (see Appendix D). This value is referred to as the *security score*. In addition, we used a binary variable called *secure* which was given if participants used at least a hash function in their *final* solutions and thus did not store the passwords in plain text.

We were also interested in participants who attempted to store user passwords securely, but struggled and then deleted their attempts from their solutions (this was coded as *attempted but failed*, or *ABF*). For this, we collected and analyzed participants' code history. In order to identify security attempts, we used the Unix *grep* utility. With grep, we searched for security-relevant terms based on the frameworks and best practices (see Appendix F). When a term was found, we analyzed the code snippets manually.

It is important to note that we still gave security scores to participants who implemented secure password storage but failed to create a functional solution, i.e., the user registration did not work. The rationale for this was that we were interested in how participants stored passwords. All other parts of the task were distraction tasks and thus of less relevance.

4. LIMITATIONS

Our study has several limitations that need to be considered when interpreting the results.

The most noteworthy limitation is that we used a convenience sample comprising 40 computer science students from a single university. Despite having a pool of 1600 computer science students at our university and offering fairly high compensation, we did not get more volunteers. We will discuss this limitation in the context of both the primary study and the meta-study. For the meta-study, we wanted a homogeneous sample so we could attribute any changes in outcome to the difference in task design. The limitation of this decision is that our results are not currently transferable to other participant groups. While we believe it is likely that other student samples will produce similar results, we expect bigger differences when working professionals are considered. It is also likely that there will be big differences between different groups of working professionals. These differences will need to be explored in future work.

The primary study is limited in the same way. Here, it would have been more desirable to have a more diverse sample; however, this would have conflicted with the need for a homogeneous sample for the meta-study. Since the meta-study was our main goal, we accepted the limitation of the primary study. That being said, there are early indications that computer science students can be acceptable proxies for professionals in developer studies [51, 38, 10, 11, 33, 14, 49, 39, 46]. This sample was not selected for its representatives and thus should not be used to infer anything about non-students.

Since our study was performed in a laboratory environment with laboratory PCs, we have an unknown amount of bias in our results. This is particularly relevant to the meta-study. While the low amount of attempted security in the non-priming condition seems plausible in light of the many password database compromises, we have no way of confirming that we are measuring the same effect. It is possible that the low amount of attempted security in the non-primed group is not due to participants' lack of awareness that passwords should be hashed and salted, but rather to a lack of concern for passwords in a study environment. In fact, we received statements to this effect in the exit survey. Of the 20 non-primed participants,

- two attempted to implement a secure solution but failed,
- two thought it was secure despite not having done anything to secure it themselves,
- two stated that they did not implement security because it was not part of the task,
- three stated that the functionality was more important to them than security,
- three were aware that security was needed but did not give any reason why they did not implement it, and
- eight were not aware that hashing and salting were important for password storage.

We must point out that the above statements are based on self-reporting by the participants. False reporting is possible in both directions. Participants who might not have been aware of the need for security might have felt embarrassed and stated that they did know but chose not to implement it and made up a reason for it. It is also possible that a participant who did know stated otherwise so as not to have to explain why security was not implemented. We must acknowledge these limitations.

We only conducted Bonferroni-Holm correction for our main hypotheses. For the rest of the exploratory analysis, we accepted the higher probability of type 1 errors to lower the risk of type 2 errors. Thus, new findings need to be confirmed by replication before they are used.

5. ETHICS

At the time of the study, our institution did not have an IRB for computer science studies. The study design was instead discussed and cleared with our independent project ethics officer. Our study also complied with the local privacy regulations. Participants gave written informed consent before participating in the study. Since half our participants underwent a deception condition, the study ended with an in-person debriefing, where all participants were informed of the true purpose of the study. Most participants were not bothered by the deception condition at all. However, some participants felt that they were judged unfairly and they stated that they would have included security if we had asked for it. After re-stating that this was completely fine, that we were interested in the APIs' ability to nudge developers toward security, and that they were not at fault, there did not seem to be any lingering negative feelings. There were also positive reactions to the deception. The majority of participants remarked that they learned a lot through the deception and will be more aware of security in future tasks and jobs, even if they are not explicitly asked to think of security.

6. HYPOTHESES & TESTS

We examined seven main hypotheses in our experiment. Two concerned the meta-focus of this paper, namely, the effect of priming/deception, denoted by P(riming). Two further concerned the A/B test comparing the two frameworks, denoted by F(ramework), and the final three were general tests concerning password storage security, denoted by G(eneral).

H-P1 Priming has an effect on the likelihood of participants attempting security.

H-P2 Priming does not have an effect on achieving a secure solution once the attempt is made.

H-F1 Framework has an effect on the security score of participants attempting security.

H-F2 Framework has an effect on the likelihood of achieving functional solutions.

H-G1 Years of Java experience have an effect on the security scores.

H-G2 If participants state that they have previously stored passwords, it affects the likelihood that they store them securely.

H-G3 Copying/pasting has an effect on the security score.

6.1 Meta-study

It is natural to assume that requesting a secure solution will lead to more attempts at security (H-P1). The interesting aspect here was how many of the non-primed participants attempted a secure solution. While we expected priming to increase the number of attempts, we did not expect a different failure rate between the priming and non-priming group (H-P2), i.e., if non-primed participants attempted security, they should not have failed more often than primed members.

6.2 Primary study

While only indirectly linked to security, we also considered the possibility that the differences between the two frameworks (JSF and Spring) could lead to different rates of functionality (H-F2). We also expected that the greater level of support offered by Spring would increase the security score of Spring participants (H-F1).

6.3 General

The above hypotheses are novel to this work. However, we also wanted to confirm findings from related studies. In their study, Acar et al. [11] observed that the programming language experience had an effect on the security of participants' solutions. Therefore, we also assumed we would observe an effect regarding experience with the Java programming language and the security score of participants' solutions (H-G1). In addition, we assumed that if participants had experience with storing user passwords in a database backend, they would be more likely to create a secure solution in the study (H-G2). Finally, several studies noted the effects of copy/paste on study results [9], especially in terms of security [10, 21, 22, 23]. Thus, we assumed that copy/paste events would affect the security code of our participants as well (H-G3).

6.4 Statistical testing

We chose the common significance level of $\alpha = 0.05$. When conducting tests on all 40 participants, we labeled the group as "all". When only testing subgroups, these were also labeled for easy interpretation. We used the Fisher's Exact Test (FET) for categorical data. For numeric data, we considered linear regression and logistic regression for binary data if the data was normally distributed. In order to test for normality, we used the Kolmogorov-Smirnov test and plotted the data for manual inspection. For data that was not normally distributed, we used the following non-parametric tests: in order to find differences between all four conditions, we used the Kruskal-Wallis test; for two groups, we used the Mann-Whitney U test. In both cases, the Levene's median-based homogeneity of variance test showed the distributions among the groups to be similar. Statistically significant values are indicated with an asterisk (*).

Of the seven main hypotheses, three concerned the security score, two concerned the binary secure value, one concerned the attempted security, and one concerned functionality. Since all but the functionality tests were closely related, we applied family-wise error correction using the Bonferroni-Holm method with a family-wise correction of 6. These analysis sections are marked with the relevant hypothesis label. However, we did not apply multiple testing correction in the exploratory part of our analysis (sections not marked with a hypothesis label). Since virtually no research has been conducted on study design for developer studies, we thought it was more important to discover interesting effects for future research to explore than it was to avoid type 1 errors while potentially dismissing an important effect merely because our sample size was not big enough (i.e., type 2 errors). For more information on family-wise errors, see [12]. To ease identification, we labelled Bonferroni-Holm corrected tests with "family = N", where N was the family size, and reported both the initial and corrected p-values.

7. RESULTS

While our main goal was analyzing the meta-study results, we began by analyzing the functionality and security of the primary study since these results were needed for the meta-analysis. Sections analyzing one of the seven main hypotheses are marked with the hypothesis label.

7.1 Functionality

Here, we discuss the functionality of the code the participants produced. We considered a solution as functional if an end-user account could be created. Figure 1 shows the distribution of functional solutions across our conditions. Of all 40 participants, 26 (65%) produced a functional solution. As shown in Figure 1, the number of participants who were able to solve the functional task is a bit higher in the Spring group compared to the JSF group.

7.1.1 Framework effects functional solution (H-F2)

Eleven of 20 (55%) participants using JSF and 15 of 20 (75%) using Spring managed to solve the task functionally. These differences were not statistically significant (sub-sample = all, FET: $p = 0.32$, odds ratio = 2.40, CI = [0.54, 11.93]). Thus, we do not reject H-F2. However, we only had a power of 0.17, so this effect is worth looking at in follow-up studies. Interestingly, a significant result would mean that the more complex framework actually has better usability with respect to functional solutions.

7.1.2 Part-time job in computer science

We asked our participants whether they had a part-time job in computer science. In prior research, Acar et al. counted students who had part-time jobs as professionals [11]. We, however, found no significant effect between having a part-time job in computer science and a functional solution (sub-sample = all, FET, $p = 1.0$, odds ratio = 0.84, CI = [0.19, 3.89]).

7.2 Security

Figure 2 shows the distribution of secure solutions across our conditions. Twelve (30%) of our 40 participants implemented some level of security for their password storage. Of the 20 participants in the non-primed groups, 0% stored the passwords securely. While we had expected significantly fewer secure solutions in the non-primed groups, we were surprised by this extreme result. From the primed group using JSF, 5 of 10 (50%) implemented some level of security (mean security score = 2.15, median = 1, sd = 2.67). From the primed group with the Spring framework, 7 of 10 (70%) participants implemented some level of security (mean security score = 4.2, median = 6.0, sd = 2.9). Table 4 shows an overview of the security scores achieved by our participants (Appendix E).

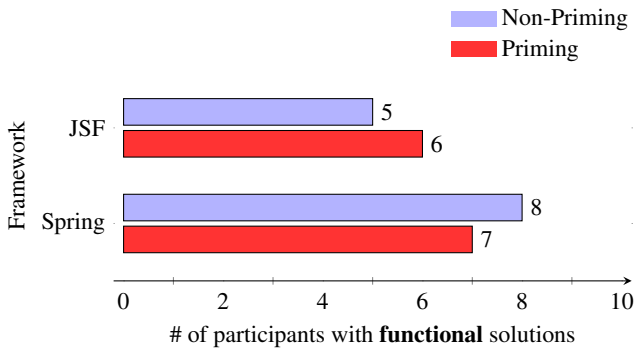

Figure 1: Functionality results per framework, split by primed vs. non-primed groups.

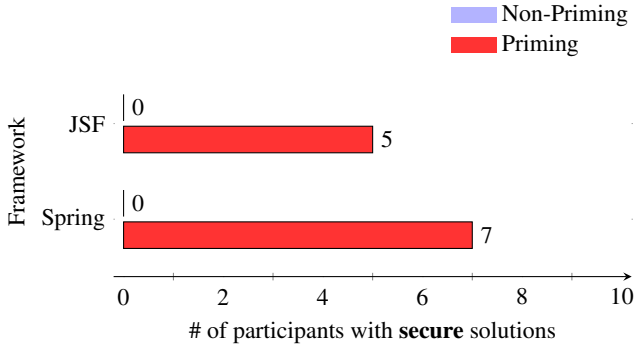

Figure 2: Security results per framework, split by primed vs. non-primed groups.

7.2.1 More Java experience, more security (H-G1)

In prior research, Acar et al. found that more Python experience leads to more security [11]. We wanted to examine this effect within our sample. We found no significant differences in our study (sub-sample = all, Kruskal-Wallis: $\chi^2 = 4.118$, $p = 0.249$, *cor-p* = 0.498, family = 6). Between the different groups of Java experience, the security score showed no significant effect. However, it must be mentioned that Acar et al. studied student and professional volunteers recruited on GitHub without compensation. Their participants had a wide range of years of experience in Python compared to our students in Java. So we have a number of differences in the samples. It is important to note this difference since it makes sense to take skills into account during condition assignment in randomized control trials. In short, we failed to confirm H-G1.

7.2.2 Previous password experience (H-G2)

We hypothesized that participants who had previous experience storing user passwords in a database backend would be more likely to add security in the study. Therefore, we wanted to test whether participants who reported having stored passwords before performed differently regarding security compared to participants who had never stored passwords before. Nine non-primed and 15 primed participants reported having stored passwords prior to the study. We found no significant differences in security in comparing the different groups of participants (sub-sample = all, FET: $p = 0.297$, *cor-p* = 0.498, odds ratio = 2.54, C.I = [0.49, 17.72], family = 6). We thus fail to reject the null hypothesis of H-G2 and cannot draw conclusions on this hypothesis. Furthermore, we calculated a power of 0.19, indicating that the effect is not reliable.

7.2.3 Framework effects security score (H-F1)

In this section, we only consider those participants who attempted security. We wanted to examine whether the framework used affected the security score (including ABF scores). We expected that Spring might score better because, in contrast to JSF, it offers built-in functions for storing passwords securely by using hashing, salting and iterations.

The descriptive statistics for the JSF group are Min 2, Median 5.5, Mean 4.3, and Max 6. The descriptive statistics for the Spring group are Min 6, Median 6, Mean 6, and Max 6. Due to the Bonferroni-Holm correction, the difference between the two groups is not flagged as significant (sub-sample = all $\wedge$ attempted security = 1 , Mann-Whitney U = 15, $p = 0.051$, *cor-p* = 0.20, family = 6). It does seem likely, though, that a larger sample would confirm the trend that Spring participants earned higher scores than JSF participants. This will be put into further context in section 7.4.1.

7.2.4 Usability of frameworks

We used the usability score from Acar et al. [9] (see Appendix C.1) to evaluate how participants perceived the usability of the two frameworks. We compared the values of the usability score for all four groups: non-primed JSF (mean = 48.25, median = 51.25, sd = 13.54), primed JSF (mean = 50.50, median = 53.75, sd = 9.78), non-primed Spring (mean = 50.50, median = 55.00, sd = 20.10), primed Spring (mean = 58.75, median = 57.50, sd = 15.65). We found no significant effect comparing all four groups (sub-sample = all, Kruskal-Wallis: $\chi^2 = 3.169$, $p = 0.37$). Furthermore, we examined whether the frameworks had different usability scores when the participants attempted to solve the task securely. We did not find a significant effect in this case either (sub-sample = all $\wedge$ attempted security = 1, Mann-Whitney U = 21.5, $p = 0.29$).

7.2.5 Security awareness

Fourteen primed and two non-primed participants believed that they managed to store user passwords securely. The two non-primed participants erroneously believed they had stored passwords securely (JN5, JN7). Their given survey answers suggested that neither had any background knowledge of password storage security at all. The primed participants were additionally asked whether they would have been aware of security if we had not explicitly ask them for it. Nine of the 14 participants indicated they would have stored user passwords securely, even if they had not been explicitly asked to do so. The fact that only two out of 20 non-primed participants attempted security suggests this is overly optimistic.

7.2.6 Security classes

In prior work, Acar et al. found that security courses had a significant effect on security [11]; therefore, we asked our participants which courses they had attended at our university in the past. We gave one point per security-relevant course. Since not all Masters students had completed their undergraduate studies at the same university, we also asked for other courses. None of the participants added a security-relevant class in the open question space. Participants reported they had attended between 0 and 4 security classes (mean = 0.8, median = 1, sd = 0.99). We found no significant evidence in the overall group (sub-sample = all, FET: $p = 0.737$, odds ratio = 1.39, CI = [0.29, 7.022]).

7.2.7 Part-time job in computer science

We found no effect between having a part-time job in computer science and a secure solution (sub-sample = all, FET, $p = 1.0$, odds ratio = 1.10, CI = [0.22, 5.30]).

7.2.8 Web browser history & task completion time

In order to analyze the web browser history, we aggregated all our participants' browser history. We assessed the visit count of all participants (mean = 179.0, median = 174.5, sd = 97.02). We could not analyze the browser history of one of our participants, because he had deleted it after completing the task. We found a total of 6224 distinct web pages for all participants. We also measured the time our participants needed to solve the task [hours] (mean = 5.11, median = 5.35, sd = 1.72). On average, participants visited 36.2 pages per hour (mean = 36.2, median = 31.52, sd = 16.44). We tested whether there was a difference in security that depended on the number of websites participants used. The results show that the website count was not significantly relevant (logistic regression, odds ratio = 1.0 , C.I = [0.99, 1.00], $p = 0.423$).

7.3 Priming

7.3.1 Priming leads to more attempts to store user passwords securely (H-P1)

The main goal of our study was to measure the effect of priming. Only two of 20 non-primed participants attempted to store the passwords securely, compared to 14 of 20 in the primed groups. This difference is statistically significant (sub-sample = all, FET: $p = 0.000$*, *cor-p* = 0.001*, odds ratio = 19.02, C.I = [3.10, 219.79], family = 6). Thus, we can reject the null of H-P1 and conclude that priming has a significant effect. We already stated we were surprised that no non-primed participant achieved a secure solution. This is mirrored in the very low number of participants who attempted to create a solution. However, we were also surprised that six participants in the primed group did not attempt a secure solution, since it was explicitly asked of them. Of these, though, three also did not manage to create a functional solution. In the exit survey, all six participants stated that they had not achieved an optimal solution and cited technical difficulties that prevented them from attempting to create a secure solution. For instance, SP6 noted: "[I] encountered errors in connecting with the DB through Spring JPA and was not able to come up with the solution. As a result [I] could not focus on implementing an algorithm to securely store the password."

It is interesting to note that even when security was explicitly stated as the goal of the study, these participants still wanted to create the functional solution before adding the security code.

7.3.2 Priming effect on achieving a secure solution once the attempt is made (H-P2)

We had hypothesized that the priming effect would only influence whether a participant would think of adding security, but once a participant had made the decision to add security, the will to follow through would be independent of priming. Now, it is very difficult to make a convincing case of no-effect using frequentist statistics with a small sample size; however, this may not be a concern. It turned out that there might actually be an effect. In the non-primed group, two of 20 attempted security but did not follow through to achieve a secure solution. In the priming group, 14 of 20 attempted and 12 achieved a secure solution. The difference between the groups is significant before correcting for multiple testing (sub-sample = all $\wedge$ attempted security = 1, FET: $p = 0.05$, *cor-p* = 0.20, odds ratio = Inf, CI = [0.64, Inf], family = 6). The same goes for the security scores (sub-sample = all $\wedge$ attempted security = 1, Mann-Whitney U: 2.0, $p = 0.034$*). Although this effect was not significant after correction, we think this is an important observation which should be examined in future studies. While it is possible that the small number of attempts in the non-primed group skewed our results, it is also possible that the failure to mention security in the task not only meant participants were not explicitly informed that security is important for password storage, but potentially discouraged participants who knew this from implementing it. This could have implications outside of study design since this effect is likely to occur in everyday life as well where developers might not be explicitly asked to secure their code and thus be dissuaded from doing so even if they know they should.

While we fail to reject the null of H-P2 due to the Bonferroni-Holm correction, we find the data to be highly interesting and suggest examining this effect in future studies.

7.4 Copy/Paste

7.4.1 Security and copy/paste (H-G3)

Our analysis of the copy/paste behavior of our participants showed another interesting result.

Of the 40 participants, only 17 copied and pasted code. Of these, 12 created a secure solution. The surprising aspect is that all secure solutions come from participants who copied and pasted security code. Not a single "non-copy/paste" participant achieved security. This difference was statistically significant (sub-sample = all, Mann-Whitney U = 57.5 $p = 0.000$*, *cor-p* = 0.000*, family = 6). Thus, we reject the null of H-G3. However, it is noteworthy that we see a positive effect of copy/paste. This is in contrast to previous work by Acar et al. [10] and Fischer et al [26]. For example, Acar et al. stated in their discussion: "Because Stack Overflow contains many insecure answers, Android developers who rely on this resource are likely to create less secure code" [10]. And Fisher et al. stated in their conclusion: "We show that 196,403 (15%) of the 1.3 million Android applications contain vulnerable code snippets that were very likely copied from Stack Overflow" [26].

These negative views are in stark contrast to our findings that 0% of participants who did not use copy/paste created a secure solution. We do not dispute the findings of Acar et al. and Fisher et al., but we do show that there is also a significant positive effect of copy/paste.

This finding also changes how we must interpret the difference in security scores between the two framework conditions presented in section 7.2.3. All secure Spring participants scored 6 points, while the JSF scores varied between 2 and 6. This could indicate that the Spring API has better usability, because it has safer defaults. However, this usability advantage seems to only affect our participants indirectly, via the web sources they use. This suggests that it is worth considering testing the usability of APIs not only with software developers but also with those who create web content. In the following section, we take a closer look at the websites used by our participants.

7.4.2 Websites used for copy/paste

Almost half of the participants (42.5%; 17/40) copied password storage examples from various websites on the Internet and pasted it to their program code. Of these, 82% (14/17) were primed participants. In all other cases, participants copied code from websites covering storage of user data in general (e.g., name, gender, email), adapting it for passwords. These websites were not considered for further analysis, since we were only interested in password storage examples.

Table 5 (Appendix F) shows all websites from which participants copied and pasted code for password storage into their solutions. The table also considers participants who attempted to store user passwords securely but did not include the security code in their final

solutions (ABF). We manually analyzed all proposed examples for password storage on these websites by using the same security scale as applied to the evaluation of participants' code (see Appendix D). If websites introduced generic solutions without predefined parameters for secure password storage, but discussed how these should be chosen in order to achieve security (e.g., OWASP: General Hashing Example (Appendix F)), we still awarded points for these parameters according to the security scale. Additionally, we compared the security scores participants received for their solutions with the scores of password storage examples from the websites they used. Since websites often contain more than one code snippet, we manually scored all of them and then used the following classification of snippets:

- **Most insecure example** - The worst solution we found on the page.
- **Obvious example** - The most obvious solution in our subjective assessment, e.g., answers on Stack Overflow that are rated with a high score by the community. For all other websites, we classified examples as obvious if they were posted at the beginning of the website.
- **Most secure example** - The solution with the highest security score.

We found that all participants who implemented password storage security (100%, 12/12) copied their program code from websites on the Internet. The majority, 75% (9/12) of participants, achieved the almost maximum score of 6/7 points in our study. These participants copied and pasted code from websites introducing up-to-date, strong algorithms. One thing the websites had in common was that all solutions had good security scores. Only one participant was on a website where the least secure example was "only" a 5.5 score. However, the most obvious example was scored with a 6 and taken by the participant.

The other three participants came across blog posts and tutorials with outdated or unsecure implementation (JP2, JP3, and JP10). For instance, JP2 copied code from a tutorial that was published in 2013 (see Appendix F, Blog Post: Hashing Example). Thus, he adopted an iteration count of 1 000 for PBKDF2, although 10 000 iterations are recommended by NIST today [30]. Interestingly, this tutorial also discussed the usage of `MD5`, `bcrypt`, and even `scrypt` with associated program code examples. The example for `MD5` was listed at the top of the website; we therefore classified it as the *obvious* example. But the author did state that this solution is vulnerable to diverse attacks and should be used with a salt. The blog post also discussed a program example for `scrypt`, which we classified as *most secure*. This was the only website visited by our participants where an example scored 7/7 points. However, JP2 decided to use `PBKDF2`, for which he found a general hashing example at the Open Web Application Security Project (OWASP) website (see Appendix F, OWASP: General Hashing Example). Although properties of parameters are discussed on the OWASP website in general, they are not applied in the code example. Therefore, JP2 searched for a similar implementation with predefined parameters and ended up with an outdated iteration count.

JP3 copied code that only contained a weak SHA1-based example. More interestingly, JP10 merged program code from four websites. Although one website included code with three points for an *obvious* example and five points for a *most-secure* example, he received only two points for his final solution. He did not use a salt, despite the fact that he copied code from an *obvious* example on Stack Overflow that considered a function with a salt as an input parameter. However, the example did not include a predefined implementation of the salt and was not implemented by our participant.

An interesting priming effect can be seen between the two participants, JP7 and SN8, who both copied code from websites in which user credentials were stored in plain text. The primed participant, JP7, used the unsecure blog post for gaining a functional solution and afterward installed a Java implementation of OpenBSD's Blowfish password hashing scheme, jBCrypt, and received six points. In contrast, the non-primed participant, SN8, did not take any further action to implement security.

Only two of the 20 non-primed participants considered security while programming, though they did not provide secure solutions in the end (JN9, SN4). JN9 was able to implement a functional solution storing user passwords securely. However, he accidentally deleted parts of his code, resulting in errors he was unable to correct. At the end, he provided a functional solution without including secure password storage. In terms of copy/paste, JN9 is interesting since the solution he implemented had, at one point, a security score of 3, although the website he used for copy/paste was scored with 2 points. He was the only participant who used a salt that he did not copy and paste from a website, but rather included it by himself. However, he used the user's email address as the salt, which is not considered a security best practice.

In summary, no participant who copied/pasted code used the *most unsecure example* on websites. Whenever the *obvious* security score differed from the *most secure* examples (true for 3/21 websites), participants used the latter. If participants' code was merged from more than one website (JP2, JP7, and JP10), participants' security score was always higher compared to the lowest-scored website, considering *most secure examples*.

7.5 Statistical testing summary

Table 2 gives an overview of the seven main hypotheses and the results of our statistical tests, with both the original and Bonferroni-Holm corrected p-values. We have two very clear results. First, concerning the meta-study: priming has a huge effect. Second, concerning the primary study: copy/paste has a strong positive effect on code security.

The effects of H-P2 and H-F1 were not statistically significant after correcting for multiple testing, but seem promising enough to examine in future work. It is also noteworthy that we did not find a significant effect for H-G1, which has been found in other studies. This is likely due to the fact that with only a student sample, the range of experience was so small that the effect is not large enough. This is important to know since it simplifies study design for developer studies conducted with students.

7.6 Examining survey open questions

We analyzed open questions of the exit survey for trends rather than for statistical significance, to gather deeper insights into the rationale behind participants' behavior.

Before mentioning security at all, we asked our participants whether they solved the task in an optimal way (see Appendix C, Q2). Thus, we were able to observe whether non-primed participants based their answers on functionality rather than on security. Seven out of 40 participants believed their solution was optimal (JP3, JN10, SN1, SN2, SN5, SN7, and SP1). In fact, most of the participants were non-primed and solved the task functionally but not securely. Some

H	Sub-sample	IV	DV	Test	O.R.	C.I.	p-value	$cor-p$-value
H-P1	-	Priming	Attempted security	FET	19.02	[3.10, 219.79]	0.000*	0.001*
H-P2	Attempted security = 1	Priming	Secure	FET	Inf	[0.64, Inf]	0.05*	0.20
H-F1	Attempted security = 1	Framework	Security score (incl. ABF)	Mann-Whitney	-	-	0.051*	0.20
H-F2	-	Framework	Functional	FET	2.40	[0.54, 11.93]	0.32	-
H-G1	-	Java experience	Security score	Kruskal-Wallis	-	-	0.249	0.498
H-G2	-	Stored passwords before	Secure	FET	2.54	[0.49, 17.72]	0.297	0.498
H-G3	-	Copy/Paste	Security score	Mann-Whitney	-	-	0.000*	0.000*

IV: Independent variable, DV: Dependent variable, O.R.: Odds ratio, C.I.: Confidence interval
Corrected with Bonferroni-Holm correction, except for H-F2.
Significant tests are marked with *.

Table 2: Summary of main hypotheses.

even stated that all requirements were functionally solved and thus their solution was optimal (JN10, SN7, SN1, SN2, and SN5). SN1, for instance, noted: "My [manually performed] tests [...] worked as expected, I should have covered everything." His answer shows that he invested some time in testing his implementation. Still, since SN1 did not think about storing the user credentials securely, it might be interesting to involve security in the testing process as well. The primed participant SP1, though, argued that his solution was optimal because the security part was sufficiently solved: "It uses `bcrypt` [with the] highest vote on [Stack Overflow link]." In contrast, a number of participants said that the quality of their code was not optimal because it did not rely on best practices, e.g., SP11: "I have probably not used best practices for Spring/Hibernate as it is the first time I used them." Other participants mentioned that exceptions and warnings need to be caught and the code can be written more cleanly and clearly (SP4, SP9, SP10, SN4, and SN9).

If participants believed they stored the user password securely, they were asked whether they solved the task in an optimal way with regard to security (see Appendix C, Q9). Only 7 of 40 participants believed that their security code was optimal (JN5, JN7, JP4, JP7, JP10, SP7, and SP11). SP7, for instance, noted that he used an "industry standard way of storing passwords" and assumed that his solution was therefore optimal. While JP3 and SP1 indicated they solved the task in an optimal way at first, they changed their minds when the question was asked in terms of security. While JP3 noted "everything is implemented", thus indicating his solution was optimal, he changed his mind with regard to security, "because the [iteration count] is not implemented yet." SP1 listed three reasons explaining why his solution is not optimal in terms of security: (1) "User is not enforced to use symbol, combination of numbers, etc.," (2) "Storing the password securely does not mean that one [person] cannot hack into another's account," and (3) "Lacking [...] 2 step validation (by phone, for example)." First, SP1 assumed that security should be implemented involving the end-user. This assumption was also made by other participants, who noticed that password validation for the end-user was missing in their solutions (SN1, SN2, SN4, SP5, JP7, SP9, and SP11). Second, SP1 did not trust password security at all, although he suggested a method for improvement (two-factor authentication). Interestingly, the non-primed participants, JN5 and JN7, indicated they stored the user password securely in an optimal way. However, we did not find any evidence of security at all, in either their solutions or in their attempts. Their answers suggested a general lack of knowledge of password storage security.

8. METHODOLOGICAL CONTRIBUTIONS

8.1 Deception

While Fahl et al. [20] found no significant difference in password studies in the behavior of end-users who were primed that the study was about passwords or received deceptive treatment, we see a very strong effect on the behavior of developers. Both design choices offer interesting insights into the problem of storing passwords securely.

If researchers wish to study the usability of a security API, priming participants is clearly the best choice, since the majority of participants in the non-primed group had no contact with the API at all and thus do not produce any data to analyze. The majority of developer user studies fall into this category.

However, these studies only look at one aspect of a much larger problem. In [21] Fahl et al. analyzed the misuse of transport layer security (TLS) APIs in Android. They found that 17% of applications using HTTPS contained dangerous code. However, 53.8% of apps did not use the TLS API at all, exposing a wealth of data to the Internet without any protection. We think it is important to study this aspect as well, and help developers become aware they need to think about security. Our results suggest that deception in studies is a promising way of studying this. It can be argued that the students simply did not include secure storage because they were in a study environment. Some participants even stated this in the exit survey and interviews. However, since there are many cases in the real world in which security is not explicitly stipulated, we think that the non-priming condition can be a valuable design for studies. This is definitively an area in which more research is needed before a reliable statement can be made.

For now, we do suggest that the usable security community also conducts developer studies using deception instead of focusing only on API use on its own. It is, however, important to conduct a full debriefing at the end to ensure the well-being of participants. In our case, we did not see any issues with the debriefing that were not addressed to the satisfaction of the participants.

8.2 Task length

The most difficult aspect of designing a deception study for developers is that distraction tasks are necessary to avoid tipping off the participants.

Short tasks Most related studies are very short [9, 10, 11, 48]. As noticed by Acar et al. [11], tasks for uncompensated developers should be designed in a way that *"participants would be likely to complete them before losing interest, but still complex enough to be interesting and allow for some mistakes."* Acar et al. [11] conducted an online experiment with 307 uncompensated GitHub users, who were asked to complete three different tasks: (1) URL shortener, (2) credential storage, and (3) string encryption. Each participant was assigned the tasks in random order. For the user credential storage task, only one function was given, which had to be completed by developers. The task was formulated in a straight forward way and

it was clear where to insert the needed code and why. Additionally, clear instructions were given to the participants, answering the question when the problem was solved. The participants were not explicitly asked to consider security. In their study, only a small number, 17.4%, stored the user passwords in plain text. A direct comparison cannot be made since the GitHub users were more experienced than the students in our study; however, the short task time and the direct instruction to store the passwords is likely to have an effect as well.

One-day time frame In contrast to tasks completed over a short time frame, longer studies are more realistic since developers have long-lasting projects and tasks they work on in the real world. In particular, it is possible to create competing requirements, pitting functionality against security in a way that is not possible in short, focused tasks. In [42], we discussed the design process of the task used in this paper in detail and how the 8 h time frame was calibrated with several pilot studies. The rational was that 8 hours is the longest time we could reasonably ask participants to remain in a lab setting. In addition, there are a number of benefits to having the participants in a controlled environment. In particular, we could fully configure the lab computers to gather a wealth of information, including full-screen capture, history of all code, copy/paste events, search history, and websites visited. Remote studies could easily use web-based editors to capture code and copy/paste events; however, gathering the rest of the information would be much more intrusive.

Multi-day time frame In a one-day time frame, we were able to conduct a task that was sufficiently long and complex that participants could perceive security as a secondary task. A multi-day time frame also offers this benefit. For instance, Bau et al. [13] investigated web application vulnerability in a multi-day experiment with eight freelancers. They were asked to develop an identity site for youth sports photo-sharing with login and different permission levels for coaches, parents, and administrators. The freelancers were primed for security by mentioning that the website "was mandated by 'legal regulations' to be 'secure', due to hosting photos of minors as well as storing sensitive contact information" [13]. The developers promised a delivery period of 35 days. Participants were compensated from three different price ranges (< $1 000, $1 000 - $2 500, and > $2 500). Two of the eight freelancers stored passwords in plain text, showing a similar distribution as in our priming condition. This design offers higher ecological validity; however, far less detailed information about the code creation process can be gathered. Both our study and the studies conducted by Acar et al. [10] have shown that information sources play a vital role in code security, which is much trickier to gather in this kind of study. So there is a trade-off between ecological validity and the ability to gather high-fidelity data.

In short, we see benefits in all three time frames and researchers now have initial data to help choose which is most appropriate for their setting.

8.3 Laboratory setting

Many developer studies are conducted online due to the difficulty of recruiting enough participants to come to a lab study. However, we found the information gathered by our instrument OS very valuable. Most developer studies contain both coding tasks and questionnaires. The questionnaires are used both for pre-screening and for gathering information on the task. While it is possible to detect the use of web sources indirectly through paste events, it is also critical to be able to detect the use of online sources during the administration of surveys.

We manually analyzed all the screen capture videos of our participants while they were answering the surveys. We could only analyze the videos of 38 participants due to technical difficulties, which meant that we were missing two videos (JN8 and SN5).[1]

We found that half the participants (20/38) used Google when answering the survey, either searching for framework-related topics (6/38) or for password storage-related topics (14/38; see Table 3). Interestingly, half the non-primed participants who did not attempt to store user passwords securely (4/8) started to search how this could be done while answering the survey. SN1, for instance, copied a survey answer from Wikipedia, explaining what hashing functions are defending against.

Of the primed participants with secure solutions, 58% (7/12) searched for additional password storage security details, e.g., in order to explain why the used algorithms were optimal or not.

Since our laboratory setting captured this information, we could take it into account during data analysis. In most online settings, this information is not available and thus there can be no certainty that the answers reflect the knowledge of the participant or just their ability to use Google.

This is particularly critical in the use of pre-screening surveys, as is done in most studies (including this one). It is common to try to screen out unsuitable candidates who do not have the technical skills needed to take part. Luckily, we only used self-assessment and reported experience to conduct the counter-balancing. However, there are also expert studies which used content-based questions for participant selection, such as the study by Krombholz et al. [38]. Here, the researchers had to be aware that a potentially large number of the participants used Google to answer the questions, which might not properly reflect their actual skills.

Being able to see all searches and information sources in direct relation to questions and answers was very valuable and is an important strength of lab-based studies. We will be releasing the study OS as an open source project, so other studies can easily capture the same information.

8.4 Qualitative vs. Quantitative study design

Finally, we want to share some observations contrasting the qualitative approach from [42] with our quantitative extension. Here, we need to distinguish between the primary study and the meta-study.

Concerning the meta-variable priming, the qualitative study already delivered a good indication that there was a significant effect, with 0 of 10 non-primed participants and 7 out of 10 primed participants achieving a secure solution. However, since small samples tend to produce more extreme results, we would not have recommended basing study design decisions on these results. With a sample size of 40 participants in the present study, we are confident this is not a fluke and that the use of deception changes the behavior of participants dramatically. It would be useful to conduct even larger studies since we currently can only expect to find large effects. However, with regard to study design, we would very much want to catch medium or even small effects as well.

For the primary study, extending the sample size allowed us to conduct an A/B test to compare two frameworks. While H-F1 was not significant in this study due to the addition of the meta-variables and consequent correction for multiple tests, even the relatively small sample size in a normal developer study would be sufficient

[1] We later discovered there was a keyboard shortcut that participants seemed to have used by accident which stopped the recording.

Group		Search	Security search
Primed	Non-Secure (6)	3	1
	ABF (2)	1	1
	Secure (12)	8	7
Non-Primed	Non-Secure (16)	7	4
	ABF (2)	1	1

Table 3: # of participants who searched on the Internet in order to fill out the survey.

to get good results. That being said, the qualitative study already highlighted many of the problems faced by developers, and the interviews were very valuable in gaining deeper insights. We did not find much to add to the conclusions of the primary study of [42] other than having stronger evidence that library support as offered by Spring has tangible benefits.

A particularly salient benefit to qualitative developer studies is that fewer participants are needed. As such, unless rigorous evidence in the context of an A/B test is needed, we think that usable security research into developers is at a stage where qualitative studies have a lot to offer and encourage the community to be more accepting of them.

9. TAKE-AWAYS

Below, we summarize the main take-aways from our study.

- Task design has a huge effect on participant behavior and deception studies seem to be a promising method for examining a previously overlooked component of developer behavior when using student participants. That said, we must reiterate important limitations to this finding. We cannot make any claims concerning studies with professionals. It seems likely that even within a group comprising professionals, there will be multiple sub-groups that will react differently under priming. This will need to be examined in future work. It is also possible that a large portion of this effect is a study artifact. In any case, we recommend more experimentation concerning the design of developer studies. Currently, researchers base task and study design mostly on gut feelings. Since we have shown that one gets vastly different outcomes, we believe it is worth investing the effort into testing multiple designs in pilot studies instead of just going with one design as is currently often the case. We also believe more effort needs to be invested in understanding what motivates developers to implement security instead of focusing too narrowly on the easier measure of API usability.

- The use of Google by participants during surveys is problematic and researchers should not rely on answers reflecting the internal knowledge of the participants. This is particularly relevant for pre-screening surveys and we strongly recommend avoiding use of answers that can be googled for participant selection or condition assignment. If at all possible, we recommend that search behavior and web usage should be tracked, because a) thus, researchers can distinguish between internal knowledge and the ability to search for knowledge; and b) seeing when and what participants google is very enlightening in itself and a valuable research instrument.

- It is our belief that qualitative research into developer behavior offers a good cost/benefit trade-off and that many valuable insights can be gained without the need for large(r) sample sizes. In addition, the use of interviews as opposed to surveys avoids the googling problem. We hope that our comparison of quantitative and qualitative examination of the same topic encourages more qualitative studies and lowers the barriers to entering into this field, since recruitment of participants is one of the biggest challenges.

- While Acar et al. have found that programming language experience has a significant effect on the security of code produced in developer studies [11], we did not find a significant effect for this. In contrast to their study, our student sample had a much smaller range of programming skills; this could explain the lack of a measurable effect. This suggests that it might not be necessary to balance programming experience when working with students, thus simplifying random condition assignment. However, our power on this test was low so this result should be replicated before it is used confidently.

- We found copy/paste has a significant positive effect on the security of our participants' code. The way previous work was set up meant that they mainly found negative effects, thus potentially skewing the perception. We think highlighting the positive side of copy/paste behavior is important.

10. CONCLUSION

In this paper, we presented an extension of our qualitative developer study on password storage [42]. The extension had the dual goal of generating insights into the effect of design for developer studies, as well as furthering the understanding of why developers struggle to store passwords securely. We examined seven main hypotheses concerning both the primary study and the meta-study. We also compared our quantitative extension to the qualitative results of [42]. Our results suggest that priming or not priming participants allows us to study different aspects of student developer behavior. Priming can be used to discover usability problems of security APIs and test improvements with a straightforward study setup. Non-priming (i.e., deception), though, might be used to research why developers do not add security without study countermeasures or being prompted. However, more work is needed to validate the ecological validity of deception in this context. We also found many participants use Google to answer survey questions. This is potentially very damaging to studies that do not account for this effect and one of many reasons we see for using qualitative research methods such as interviews to study developers.

The next step in this research endeavor is designing an experiment to study the priming effect with professionals. Since it is unrealistic to expect even a small number of working professionals to sacrifice a full day to take part in a lab study, a different study design will be needed. We also plan to study additional design variables for developer studies to create a stronger foundation for conducting usable security and privacy research with professionals.

11. ACKNOWLEDGMENTS

This work was partially funded by the ERC Grant 678341: Frontiers of Usable Security.

12. REFERENCES

[1] Githut: A small place to discover languages in github. http://githut.info/, February 6, 2018 visited.

[2] Glipper is a clipboardmanager for gnome. https://launchpad.net/glipper, February 6, 2018 visited.

[3] Pypl popularity of programming language. http://pypl.github.io/PYPL.html, February 6, 2018 visited.

[4] The redmonk programming language rankings. http://redmonk.com/sogrady/2017/06/08/language-rankings-6-17/, February 6, 2018 visited.

[5] Tiobe index. http://www.tiobe.com/tiobe-index/, February 6, 2018 visited.

[6] Trendy skills: Extracting skills that employers seek in the it industry. http://trendyskills.com/, February 6, 2018 visited.

[7] W3techs web technology surveys: 'usage of server-side programming languages for websites'. https://w3techs.com/technologies/overview/programming_language/all, February 6, 2018 visited.

[8] R. Abu-Salma, M. A. Sasse, J. Bonneau, A. Danilova, A. Naiakshina, and M. Smith. Obstacles to the adoption of secure communication tools. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 137–153. IEEE, 2017.

[9] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. Comparing the usability of cryptographic apis. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*, 2017.

[10] 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 *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 289–305. IEEE, 2016.

[11] Y. Acar, C. Stransky, D. Wermke, M. L. Mazurek, and S. Fahl. Security developer studies with github users: Exploring a convenience sample. In *Symposium on Usable Privacy and Security (SOUPS)*, 2017.

[12] R. A. Armstrong. When to use the Bonferroni correction. 34:502–508, 2014.

[13] J. Bau, F. Wang, E. Bursztein, P. Mutchler, and J. C. Mitchell. Vulnerability factors in new web applications: Audit tools, developer selection & languages. *Stanford, Tech. Rep*, 2012.

[14] P. Berander. Using students as subjects in requirements prioritization. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pages 167–176. IEEE, 2004.

[15] J. Bonneau and S. Preibusch. The password thicket: Technical and market failures in human authentication on the web. In *WEIS*, 2010.

[16] J. Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[17] S. Clarke. Using the cognitive dimensions framework to design usable apis.

[18] J. Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[19] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[20] S. Fahl, M. Harbach, Y. Acar, and M. Smith. On the ecological validity of a password study. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, page 13. ACM, 2013.

[21] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[22] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith. Hey, you, get off of my clipboard. In *International Conference on Financial Cryptography and Data Security*, pages 144–161. Springer, 2013.

[23] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith. Rethinking ssl development in an appified world. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 49–60. ACM, 2013.

[24] M. Finifter and D. Wagner. Exploring the relationship between web application development tools and security. In *USENIX conference on Web application development*, 2011.

[25] K. Finstad. Response interpolation and scale sensitivity: Evidence against 5-point scales. *Journal of Usability Studies*, 5(3):104–110, 2010.

[26] 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. CoRR abs/1710.03135*, 2017.

[27] J. L. Fleiss, B. Levin, and M. C. Paik. *Statistical methods for rates and proportions*. John Wiley & Sons, 2013.

[28] A. Forget, S. Chiasson, P. C. Van Oorschot, and R. Biddle. Improving Text Passwords Through Persuasion. In *Proceedings of the 4th Symposium on Usable Privacy and Security*, pages 1–12. ACM, jul 2008.

[29] P. Gorski and L. L. Iacono. Towards the usability evaluation of security apis. In *Proceedings of the Tenth International Symposium on Human Aspects of Information Security & Assurance (HAISA 2016)*, page 252. Lulu. com, 2016.

[30] P. A. Grassi, E. M. Newton, R. A. Perlner, A. R. Regenscheid, W. E. Burr, J. P. Richer, N. B. Lefkovitz, J. M. Danker, Y.-Y. Choong, K. Greene, et al. Digital identity guidelines: Authentication and lifecycle management. *Special Publication (NIST SP)-800-63B*, 2017.

[31] M. Green and M. Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[32] S. M. T. Haque, M. Wright, and S. Scielzo. A Study of User Password Strategy for Multiple Accounts. pages 1–3.

[33] 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.

[34] I. Ion, R. Reeder, and S. Consolvo. "... no one can hack my mind": Comparing expert and non-expert security practices. In *SOUPS*, volume 15, pages 1–20, 2015.

[35] B. Kaliski. Pkcs# 5: Password-based cryptography specification version 2.0, Sept. 2000.

[36] A. J. Kimmel. *Ethical issues in behavioral research: Basic and applied perspectives*. John Wiley & Sons, 2009.

[37] R. E. Kirk. *Experimental design*. Wiley Online Library, 1982.

[38] K. Krombholz, W. Mayer, M. Schmiedecker, and E. Weippl. "I Have No Idea What I'm Doing" âĂŞ On the Usability of Deploying HTTPS. *USENIX Security*, pages 1–18, jun 2017.

[39] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental

models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM, 2006.

[40] D. Lazar, H. Chen, X. Wang, and N. Zeldovich. Why does cryptographic software fail?: a case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 7. ACM, 2014.

[41] S. Nadi, S. Krüger, M. Mezini, and E. Bodden. Jumping through hoops: why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering*, pages 935–946. ACM, 2016.

[42] A. Naiakshina, A. Danilova, C. Tiefenau, M. Herzog, S. Dechand, and M. Smith. Why do developers get password storage wrong?: A qualitative usability study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 311–328. ACM, 2017.

[43] J. Nielsen. *Usability engineering*. Elsevier, 1994.

[44] L. Prechelt. Plat_forms: A web development platform comparison by an exploratory experiment searching for emergent platform properties. *IEEE Transactions on Software Engineering*, 37(1):95–108, 2011.

[45] N. Provos and D. Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.

[46] I. Salman, A. T. Misirli, and N. Juristo. Are students representatives of professionals in software engineering experiments? In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 666–676. IEEE Press, 2015.

[47] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor's new security indicators. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 51–65. IEEE, 2007.

[48] J. Stylos and B. A. Myers. The implications of method placement on api learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 105–112. ACM, 2008.

[49] M. Svahnberg, A. Aurum, and C. Wohlin. *Using students as subjects - an empirical evaluation*. ACM, New York, New York, USA, Oct. 2008.

[50] R. Wash and E. Rader. Influencing mental models of security: a research agenda. In *Proceedings of the 2011 New Security Paradigms Workshop*, pages 57–66. ACM, 2011.

[51] 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 *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 158–177. IEEE, 2016.

APPENDIX

We used a seven-point rating scale according to [25].

A. PRE-SCREENING QUESTIONNAIRE

1. Gender: *Female/Male/Other/Prefer not to say*
2. Which university are you at? *University of Bonn/Other: [free text]*
3. In which program are you currently enrolled? *Bachelor Computer Science/Master Computer Science/Other: [free text]*
4. Your semester: [free text]
5. How familiar are you with Java?
   1 - Not familiar at all - 7 - Very familiar
6. How familiar are you with PostgreSQL?
   1 - Not familiar at all - 7 - Very familiar
7. How familiar are you with Hibernate?
   1 - Not familiar at all - 7 - Very familiar
8. How familiar are you with Eclipse IDE?
   1 -Not familiar at all - 7 - Very familiar

B. ENTRY SURVEY

Before solving the task, participants were asked questions Q5 - Q8 from the pre-screening questionnaire (Appendix A) one more time for consistency reasons. Additionally, they were asked two further questions:

1. *Expectation*:
   What is your expectation? Overall, this task is
   1 - Very difficult - 7 - Very easy
2. How familiar are you with JavaServer Faces (JSF)/Spring?
   1 - Not familiar at all - 7 - Very familiar

C. EXIT SURVEY

Questions asked after solving the task:

1. *Experience*
   Overall, this task was
   1 - Very difficult - 7 - Very easy
2. Do you think your solution is optimal? *No / Yes*
   - Why do you think your solution is (not) optimal? [free text]
3. I have a good understanding of security concepts.
   1 - Strongly disagree - 7- Strongly agree
4. How often do you ask for help facing security problems?
   1- Never - 7 - Every time
5. How often are you asked for help when somebody is facing security problems?
   1- Never - 7 - Every time
6. How often do you need to add security to the software you develop in general (Primed group: apart from this study)?
   1- Never - 7 - Every time
7. How often have you stored passwords in the software you have developed (Primed group: apart from this study)?
8. How would you rate your background/knowledge with regard to secure password storage in a database?
   1- Not knowledgeable at all - 7 Very knowledgeable
9. Do you think that you stored the end-user passwords securely? *No / Yes*
   - If Yes:
     - What did you do to store the passwords securely? [free text]
     - Do you think your solution is optimal? *No / Yes*
       - Why do you think your solution is (not) optimal? [free text]
     - Primed group: Do you think you would have stored end-user passwords securely, if you had not been told about it? Please explain your decision. [free text]
   - If No:
     - Why do you think that you did not store the passwords securely? [free text]

  - Non-Primed group: Were you aware that the task needed a secure solution? *No / Yes*
  - What would you do, if you needed to store the end-user passwords securely? [free text]

10. Did you use libraries to store the end-user passwords securely? *No / Yes*
    - If Yes:
      - Which libraries did you use to store the end-user passwords securely (in this study)? [free text]
      - Please name the most relevant library you have used to store the end-user passwords securely (in this study). [free text]
      - You have identified {*participant's answer*} as the most relevant library to store end-user passwords securely. How would you rate its ease of use in terms of accomplishing your tasks functionally / securely? *1- Very Difficult - 7- Very Easy*
        Please explain your decision. [free text]
      - Usability scale for {*participant's answer*} (see C.1)
11. JSF/ Spring supported me in storing the end-user password securely. *1 - Strongly disagree - 7- Strongly agree*
    Please explain your decision. [free text]
12. JSF/ Spring prevented me in storing the end-user password securely. *1 - Strongly disagree - 7- Strongly agree*
    Please explain your decision. [free text]
13. JSF/ Spring: Usability scale (see C.1); the term *library* was replaced by *framework*.
14. Have you used Java APIs / libraries to store end-user passwords securely before? *No / Yes*
    - If Yes:
      - Which Java APIs / libraries to store end-user passwords securely have you used before? [free text]
      - What is your most-used API / library for secure password storage? [free text]
      - How would you rate its ease of use in terms of accomplishing your tasks functionally? *1- Very Difficult - 7- Very Easy*
        Please explain your decision. [free text]
      - How would you rate its ease of use in terms of accomplishing your tasks securely? *1- Very Difficult - 7- Very Easy*
        Please explain your decision. [free text]

C.1 Usability scale from [9]

By contrast to [9] we dropped the option "does not apply" for the last two questions, Q10 and Q11. Used scale in our study:

Please rate your agreement to the following questions on a scale from 'strongly agree' to 'strongly disagree.' (Strongly agree; agree; neutral; disagree; strongly disagree). Calculate the 0-100 score as follows: 2.5 * (5-Q_1 + $\sum_{i=2..10}$ (Q_i-1)); for the score, Q11 is omitted.

- 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 task.
- 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.

Please rate your agreement to the following questions on a scale from 'strongly agree' to 'strongly disagree'. (Strongly agree; agree; neutral; disagree; strongly disagree).

- 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.

C.2 Demographics

- Please select your gender. *Female/Male/Other/Prefer not to say*
- Age: [free text]
- What is your current occupation? *Student Undergraduate/Student Graduate/Other: [free text]*
- At which university are you currently enrolled? *University of Bonn / University of Aachen*
- Which security lectures did you pass in your Bachelor/Master programme? *(To select)/Other: [free text]*
- Currently, do you have a part-time job in the field of Computer Science? If yes, please specify: [free text]
- How many years of experience do you have with Java development? *< 1 year/ 1 - 2 years/ 3 - 5 years/ 6 - 10 years/ 11+ year*
- What is your nationality? [free text]
- Thank you for answering the questions! If you have any comments or suggestions, please leave them here: [free text]

D. SECURITY SCORE

We used the following security score from Naiakshina et al. [42] for the evaluation of participants' solutions:

1. The end-user password is salted (+1) and hashed (+1).
2. The derived length of the hash is at least 160 bits long (+1).
3. The iteration count for key stretching is at least 1 000 (+0.5) or 10 000 (+1) for `PBKDF2` [35] and at least $2^{10} = 1\,024$ for `bcrypt` [45] (+1).
4. A memory-hard hashing function is used (+1).
5. The salt value is generated randomly (+1).
6. The salt is at least 32 bits in length (+1).

E. SECURITY RESULTS

Table 4 summarizes the security evaluation for participants' implemented solutions as introduced in [42] with slightly modifications, e.g., the digest size of `bcrypt` was changed from 192 bits to 184 bits, reasonable by practical implementation standards. Table 4 also considers participants who attempted to store end-user passwords securely during programming, but removed the security code from their final solutions (ABF = attempted but failed).

	Time	Functionality	Security					
			Hashing			Salt		
	(hh:mm)	Storage working	Hashing function (at most +2)	Digest size (bits) (+1 if $\geq$ 160)	Iteration count (at most +1)	Generation (at most +2)	Length (bits) (+1 if $\geq$ 32)	**Total** (7)
JN2*	04:05	✓						
JN3*	03:01	✓						
JN4*	04:11	✗						
JN5*	05:30	✗						
JN6	05:13	✓						
JN7	07:33	✗						
JN8	07:33	✗						
JN9	06:08	✓	SHA1	160	1	end-user email address	8	3 (ABF)
JN10	03:45	✓						
JN11	06:36	✗						
JP1*	04:55	✓						
JP2*	03:12	✓	PBKDF2(SHA256)	512	1000	SecureRandom	256	5.5
JP3*	05:29	✓	SHA256	256	1			2
JP4*	04:12	✓	PBKDF2(SHA1)	160	20000	SecureRandom	64	6
JP5*	06:32	✓						
JP6	07:33	✗						
JP7	06:08	✗	BCrypt	184	2^{12}	SecureRandom	128	6
JP8	07:22	✗						
JP9	07:18	✗	BCryp	184	2^{8}	pgcrypto	128	5 (ABF)
JP10	04:45	✓	SHA256	256	1			2
SN1*	03:15	✓						
SN2*	02:24	✓						
SN3*	02:01	✓						
SN4*	04:01	✓	BCrypt	184	2^{10}	SecureRandom	128	6 (ABF)
SN5*	04:50	✓						
SN6	07:03	✗						
SN7	05:35	✓						
SN8	07:33	✗						
SN9	05:31	✓						
SN10	03:23	✓						
SP1*	03:15	✓	BCrypt	184	2^{10}	SecureRandom	128	6
SP3*	07:00	✗	BCrypt	184	2^{10}	SecureRandom	128	6
SP4*	03:39	✓	BCrypt	184	2^{10}	SecureRandom	128	6
SP5*	03:44	✗						
SP6	07:33	✓						
SP7	01:49	✓	BCrypt	184	2^{11}	SecureRandom	128	6
SP8	05:59	✗	#					0 (ABF)
SP9	05:50	✓	BCrypt	184	2^{10}	SecureRandom	128	6
SP10	05:53	✓	BCrypt	184	2^{10}	SecureRandom	128	6
SP11	03:15	✓	BCrypt	184	2^{10}	SecureRandom	128	6

Table 4: Password security evaluation, including participants who attempted to implement security but failed (ABF).
Labeling of participants: S = Spring, J = JSF, P = Priming, N = Non-Priming
* = Used for the qualitative study in [42].
# = Used Spring Security's `PasswordEncoder` interface without deciding for an algorithm.

F. COPY/PASTE WEBSITES

Table 5 lists all websites used by participants who implemented user password storage security. We also examined websites used by participants who attempted to store passwords securely, but removed all security-relevant code from their solutions (ABF = attempted but failed). In order to search for programming security attempts we used the Unix utility *grep*. The following search words were used for security attempt identification: encode, sha, pbkdf2, scrypt, hashpw, salt, MD5, passwordencoder, iterations, pbekeyspec, argon2, bcrypt, messagedigest, crypt.

Participant	Security score	Website	Description	Most insecure example	Obvious example	Most secure example
JN9	3 (ABF)	www.sha1-online.com/sha1-java/	Blog Post: SHA1 Java	2	2	2
JP2	5.5	https://www.owasp.org/index.php/Hashing_Java	OWASP: General Hashing Example	6	6	6
		https://stackoverflow.com/questions/18268502/how-to-generate-salt-value-in-java	Stack Overflow: Salt Example	3	3	3
		https://howtodoinjava.com/security/how-to-generate-secure-password-hash-md5-sha-pbkdf2-bcrypt-examples/#PBKDF2WithHmacSHA1	Blog Post: Hashing Example	1	1	7
JP3	2	www.mkyong.com/java/java-sha-hashing-example/	Blog Post: Hashing Example	2	2	2
JP4	6	http://blog.jerryorr.com/2012/05/secure-password-storage-lots-of-donts.html	Blog Post: Hashing Example	5.5	6	6
JP7	6	http://javaandj2eetutor.blogspot.de/2014/01/jsf-login-and-register-application.html	Blog Post: Hashing Example	0	0	0
		www.mindrot.org/projects/jBCrypt/	Documentation: Java Implementation jBCrypt	6	6	6
JP9	5.5 (ABF)	https://www.meetspaceapp.com/2016/04/12/passwords-postgresql-pgcrypto.html	Blog Post: Hashed Passwords with PostgreSQL's pgcrypto	5.5	5.5	5.5
JP10	2	https://stackoverflow.com/questions/33085493/hash-a-password-with-sha-512-in-java	Stack Overflow: Hashing Example	3	3	5
		https://stackoverflow.com/questions/3103652/hash-string-via-sha-256-in-java	Stack Overflow: Hashing Example	2	2	2
		https://docs.oracle.com/javase/7/docs/api/java/security/MessageDigest.html	Documentation: Class MessageDigest	1	2	2
		https://stackoverflow.com/questions/11665360/convert-md5-into-string-in-java	Stack Overflow: Convert MD5 into String in Java	1	1	1
SN4	6 (ABF)	http://websystique.com/spring-security/spring-security-4-password-encoder-bcrypt-example-with-hibernate/	Blog Post: Hashing Example	6	6	6
SN8	0	https://dzone.com/articles/spring-mvc-example-for-user-registration-and-login-1	Blog Post: Hashing Example	0	0	0
SP1	6	https://stackoverflow.com/questions/25844419/spring-bcryptpasswordencoder-generate-different-password-for-same-input	Stack Overflow: Hashing Example	6	6	6
SP3, SP4, SP11	6	www.mkyong.com/spring-security/spring-security-password-hashing-example/	Blog Post: Hashing Example	6	6	6
SP7	6	https://hellokoding.com/registration-and-login-example-with-spring-xml-configuration-maven-jsp-and-mysql/	Blog Post: Hashing Example	6	6	6
SP8	0 (ABF)	www.websystique.com/springmvc/spring-mvc-4-and-spring-security-4-integration-example/	Blog Post: Hashing Example	6	6	6
SP9	6	https://stackoverflow.com/questions/18653294/how-to-correctly-encode-password-using-shapasswordencoder	Stack Overflow: Hashing Example	5.5	6	6
SP10	6	https://stackoverflow.com/questions/42431208/password-encryption-in-spring-mvc	Stack Overflow: Password Encryption in Spring MVC	6	6	6

Table 5: Websites from which participants copied and pasted code for password storage.