Securing Your Crypto-API Usage Through Tool Support - A Usability Study

omitted for submission

Abstract—**Developing secure software is essential for protecting passwords and other sensitive data. Despite the abundance of cryptographic libraries available to developers, prior work has shown that developers often unknowingly misuse the provided Application Programming Interfaces (APIs), resulting in serious security vulnerabilities. Eclipse CogniCrypt is an IDE plugin that aims at helping developers use cryptographic APIs more easily and securely by providing three main functionalities: (1) it provides a use-case-oriented view of cryptographic APIs and guides the developer through their configuration, (2) it generates the code needed to accomplish the chosen use case based on the selected choices, and (3) it continuously analyzes the developer's code to ensure that no API misuses are introduced later. However, so far the effectiveness of CogniCrypt was never empirically evaluated. In this work, we fill this gap through a controlled experiment with 24 Java developers. We evaluate the tool's effectiveness in reducing API misuses and saving developer time. The results show that CogniCrypt significantly improves code security and also speeds up development for cryptography-related tasks. The feedback received during the study suggests that developers particularly appreciate CogniCrypt's code generation. Its static-analysis is valued for keeping the code up-to-date. Yet, the further integration of generated code into a developer's project still presents a major challenge. Nonetheless, our results show that CogniCrypt effectively helps application developers produce more secure code.**

Index Terms—**component, formatting, style, styling, insert**

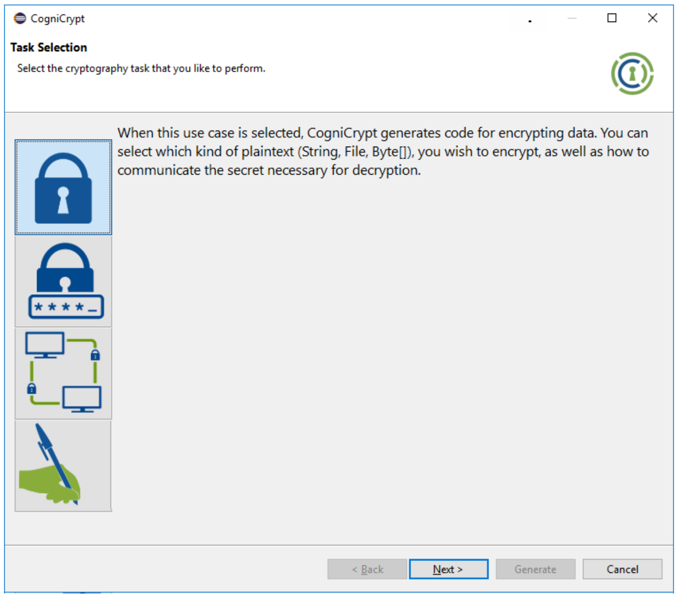
I. Introduction

Entrusting sensitive data such as credit-card information or passwords to digital devices has become the norm. To secure this data, application developers need to leverage cryptographic algorithms. As a result, a multitude of cryptographic Application Programming Interfaces (APIs) have been developed for most of the major programming languages over the years. These APIs provide functionalities such as digital signatures, encryption algorithms, and hashing functions that can provide the required protection. Unfortunately, previous research has shown that many application developers do not correctly use these cryptographic APIs, leading to serious security vulnerabilities [5, 14, 28] such as leaking passwords or insecure encryptions.

Previous work investigated why developers struggle with cryptography APIs and what solutions they deem fit to solve their problems [19]. The authors concluded that developers desire higher-level abstractions in terms of tasks. These abstractions can be in terms of more use-case-based documentation, use-case-based API design, or tooling that helps them write better code for these use cases and alerts them of any problems. Additionally, developers desire support for selecting the correct cryptographic algorithms and configurations. In response, Krüger et al. [13] proposed CogniCrypt, an Eclipse-based assistant for cryptogtaphic APIs, that addresses some of these problems. CogniCrypt is targeted at application developers who need to use cryptographic APIs but are not necessarily cryptography experts. In particular, CogniCrypt aids developers by providing a code generator for several cryptographic use cases as well as a suite of code analyses guaranteeing the correct usage of common cryptographic APIs. Krüger et al. [13, 15, 16] argue that this mix of code generation and code analysis leads developers to select the right solutions for their projects as well as make sure the implementation is correct. However, these claims have not been empirically investigated, leaving the question whether CogniCrypt may reduce cryptographic misuse unanswered.

In this paper, we bridge this gap by conducting a controlled experiment to evaluate the usefulness of such an assistance tool from the developer's perspective. Our experiment involves 24 participants from two universities. We employ a within-subjects design [11] where each participant performs two tasks, one using CogniCrypt and one using the plain Eclipse IDE. Our evaluation focuses on answering the following research questions:

- RQ_1 *Does the use of CogniCrypt improve the functional correctness of cryptography application code?* We are interested to see if participants who use CogniCrypt end up producing more functional code for a given task. We measure functionality by manually assessing participants' code for functionality based on a functionality scoresheet we developed.
- RQ_2 *Does the use of CogniCrypt improve the security of cryptography application code?* Given the main claims of CogniCrypt, we are interested to see if participants who use CogniCrypt do end up producing more secure code. We measure security in terms of how many cryptographic API misuses they make.
- RQ_3 *Does the use of CogniCrypt shorten the time taken to write cryptography application code?* Given its task-based nature, CogniCrypt is supposed to save the time needed to research and understand the various details of cryptography APIs. We are interested to see if this indeed holds. Do participants using CogniCrypt end up finishing the cryptography tasks faster?
- RQ_4 *Do developers perceive CogniCrypt to be more usable*



When this use case is selected, CogniCrypt generates code for encrypting data. You can select which kind of plaintext (String, File, Byte[]), you wish to encrypt, as well as how to communicate the secret necessary for decryption.

Fig. 1: CogniCrypt's selection screen.

than plain Eclipse? Since the usability of any tool impacts its long-term adoption, we are also interested to evaluate participants' perception of CogniCrypt. We measure usability through the Net Promoter Score (NPS) [27] and direct written feedback by participants.

RQ_5 *What obstacles do developers still face with CogniCrypt?* When users face roadblocks while using a tool, they might stop using it, even if they consider it otherwise usable. From written feedback, we therefore also derive obstacles participants report.

Our results show that there is a statistically significant improvement in functionality and security scores as well as completion times for participants using CogniCrypt over regular Eclipse. Moreover, the majority of participants using Eclipse could not finish the given tasks in the allotted time, while more than 80% of participants using CogniCrypt finished their given task. Overall, the results provide strong evidence that CogniCrypt can indeed help developers write more secure code faster.

II. CogniCrypt

CogniCrypt [13, 14] is an assistance tool for cryptographic APIs. It has been integrated with the widely used integrated development environment (IDE) Eclipse [31]. Its main features are a code generator CogniCrypt$_{GEN}$ and a static analysis CogniCrypt$_{SAST}$. We only give a brief description of the tool and refer to the tool paper by Krüger et al. [13] for a more detailed discussion of features and supported use cases.

The code generator CogniCrypt$_{GEN}$ supports four common use cases of cryptography as the screenshot in Figure 1 shows. These include data encryption and establishing a TLS communication channel. The CogniCrypt code generator uses a wizard to guide the developer through the configuration process of the generated code. First, a user can select the use case they wish to perform from the four pre-defined use cases. For CogniCrypt$_{GEN}$ to configure the correct solution, it may ask the user several questions (e.g., data type of plain text or whether the server or client side is implemented). The questions are intended to require little cryptography knowledge, yet still allow CogniCrypt$_{GEN}$ to tailor the solution to the user's use case. As a last step before CogniCrypt$_{GEN}$ generates the code into the user's project, they have to select a file. This file is used by CogniCrypt$_{GEN}$ to place a new method that showcases how to make use of the actual implementation of the use case. This method simplifies the integration of CogniCrypt$_{GEN}$'s code into the remainder of the user's project. The actual implementation code comprises wrapper code around existing widely used Java cryptographic APIs (e.g., Java Cryptography Architecture [25], Java Secure Socket Extension [26]) a developer may use to implement these use cases themselves. CogniCrypt$_{GEN}$ generates this code into separate classes into the package 'cognicrypt.crypto'. To support this process further and also the evolution, modification, and refactoring of the generated code by the user, CogniCrypt also comprises a static misuse detector CogniCrypt$_{SAST}$. By default, CogniCrypt$_{SAST}$ is applied every time the user saves a source file. However, they may also disable automated execution and trigger it through a button in the toolbar or via the project context menu of the package explorer. For all cases, CogniCrypt shows misuses as regular Eclipse warnings. In a CogniCrypt preference menu, the user can also select the severity of warnings based on the type of misuse.

In the backend, both CogniCrypt$_{GEN}$ and CogniCrypt$_{SAST}$ make use of the specification language CrySL [14]. When bootstrapped with a set of CrySL rules, CogniCrypt$_{SAST}$ the developer's code on the fly in Eclipse for its compliance with the constraints on parameter values, forbidden methods, and usage patterns defined in them. CogniCrypt$_{GEN}$, on the other hand, uses use-case-specific Java code templates with gaps where code using Crypto APIs in a full implementation of the respective use case would be. To generate the full implementation, CogniCrypt$_{GEN}$ applies the appropriate CrySL rules to the right template to fill the gaps [16].

III. Experimental Design

We next describe the controlled experiment we designed to address this study's goal.

A. *Object of the Experiment and Methodology*

To measure CogniCrypt's effectiveness and answer our five research questions, we compare the cryptography code software developers write with and without CogniCrypt. To this end, we designed the experiment such that each participant is asked to implement two programming tasks that involve cryptography. For one of them, they are allowed to use CogniCrypt, for the other one they use a regular Eclipse. We compare against a regular Eclipse to most closely resemble an everyday working environment of application developers.

TABLE I: Study Tasks for Participants

Name	Goal	Program Stub	Test Cases
FE	Encrypt a file	Reads file and writes it back to disk	write of ciphertext-file was successful ciphertext-text file existence ciphertext file is not empty ciphertext file is not same as plaintext-file
TLS	Send specific message to a server via TLS connection	Message that should be sent is defined	correct message incorrect message (4x)

In the following, we will refer to the environments as "CC" (for **C**ogni**C**rypt) and "EC" (for **EC**lipse).

We follow a *within-subjects* design to ensure that we can observe the effect of CogniCrypt per participant and avoid possible biases or population differences caused by the distribution of participants among two separate groups [3]. A within-subjects design allows us to run the experiment with a smaller number of participants than would have been needed for a between-subjects design. It is also resilient towards variability in individual skill level since it compares scores of one participant in one condition with the scores of the same participant in a different condition. This design further provides a better chance of observing any statistical differences between the two tested environments EC and CC. To avoid learning/practice effects as well as fatigue effects that might influence the solutions, we follow a Latin square design [7] where the order of the tasks and environments presented to the participants is assigned in a way such that each task appears in each sequential position an equal number of times. In other words, an equal number of participants receive each possible ordering of tasks and environments.

Before each task, we ask participants to read through a tutorial consisting of a handful of lines of text and some screenshots on the environment they would be using in the next task. The experiment instructor asks them to make use of the features mentioned and explained in the tutorial as much as possible while working on the task. We have, however, avoided providing any particular in-depth documentation on CogniCrypt$_{GEN}$ and CogniCrypt$_{SAST}$. This decision—if anything—puts CogniCrypt at a disadvantage as participants are more likely to be familiar with regular Eclipse than with CogniCrypt and every tool comes with a learning curve. However, we did not want to unnecessarily bias participants since we believe that the more documentation we were to provide, the clearer it would be which of the two tools was the one we were evaluating. Such bias would severely limit the value of the feedback participants give us on CogniCrypt's feedback in the survey.

While solving the tasks, participants are allowed to use any online resources they want to, apart from email and chat applications. We also prime participants by enhancing task descriptions with requests to participants to pay extra attention to security while implementing the task. The rationale is that previous research strongly suggests that developers, in the context of user studies, do not bother with security concerns unless explicitly requested [22].

We design the two tasks shown in Table I. For the tasks, we implemented two small Java program stubs (involving 1–3 classes) that participants had to modify during the experiment. For each task, participants need to add certain security functionality to the existing program stub. Task FE requires the participant to implement a secure file encryption using a password. The program stub we provided reads the file into a string and then stores that string into a file again. In task TLS, we expect participants to implement a TLS client whose server runs locally on their machine at port 9999. For this task, the stub defines the message that should be sent. It also contains a key-store file that stores the certificate the TLS connections must use when connecting to the server. The task description pointed participants to this file.

With each stub, we provide several unit tests, each covering one requirement for functional correctness. The task descriptions explain that a task is completed once all unit tests pass. They also point participants to the exact method stubs to implement, such that they can run the unit test before submitting their code. We have further enhanced the program stubs with todo-comments at the program locations that require participants' extensions.

When participants use a bare Eclipse, they are required to write code that implements the task according to security and functionality criteria we define below and integrate that code into the stub. When participants may use CogniCrypt, the tool can generate code that implements the criteria for them. Here, the challenge is no longer about interacting with Crypto APIs directly and instead becomes about selecting the right task in CogniCrypt$_{GEN}$, answering its configuration questions correctly and integrating the `templateUsage()` into the provided stub. There are several correct configurations of CogniCrypt for both tasks and, consequently, multiple different correct `templateUsage()` for both tasks, too. If configured correctly, the `templateUsage()`, as shown in Figure 2, for task FE would take a password and String plain text and contain three three method calls that subsequently generate a key from the password, encrypt the plaintext using the key, and decrypt the resulting ciphertext with the same key. In case of the TLS task, one possible correct method `templateUsage()` is shown in Figure 3. The method takes the host and port and first instantiates an object of another generated class, which handles TLS connections. Subsequently, it calls three methods on this object, one to send data, one to receive data and one to close the

TABLE II: Functionality and Security Criteria for Study Tasks

Name	Functionality	Security
FE	Writing of ciphertext file was successful (test) Ciphertext file existence (test) Ciphertext file is not empty (test) Ciphertext file is not equal to plaintext file (test) Password used for key generation Using some kind of encryption Encrypting the whole plaintext	Using secure encryption configuration Using secure key deriviation Password has never been a String Using secure hashing algorithm for key derivation Random salt of at least 16 byte Secure preparation of encryption
TLS	Correct message (test) 4x Incorrect Message (test) Using provided parameters Setting correct key store Flushing of write channel Closing Connection	Using TLS Using secure SSL socket factory provider Using secure cipher suites Using secure tls protocols

connection.

```
public static boolean templateUsage(String plainText, char[]
 encryptionPassword) throws GeneralSecurityException {

    SecureEncryptor encryptor = new SecureEncryptor();
    SecretKey key = encryptor.generateKey(encryptionPassword);

    String ciphertext = encryptor.encrypt(plaintext, key);
    encrypt.decrypt(ciphertext, key);
}
```

Fig. 2: TemplateUsage Method for ENC task

```
public static void templateUsage(String host, int port) {
    //You need to set the right host (first parameter) and the
      port name (second parameter). If you wish to pass an IP
      address, please use overload with InetAdress as second
      parameter instead of string.

    TLSClient tls = new TLSClient(host, port);
    boolean sendingSuccessful = tls.sendData(""); // This call
      sends the passed message over the connection.

    String data = tls.receiveData(); //This call makes the
    socket listen for incoming messages.

    tls.closeConnection(); // This call properly closes the
      connection. Do not forget it.
}
```

Fig. 3: TemplateUsage Method for TLS task

The two environments reflect realistic development settings that, as pointed out above, each come with their own challenges. When developers may not use CogniCrypt, they have to gather the domain and API-usage knowledge for Crypto API themselves. When they may use the tool, they still need to use it correctly and integrate the generated code properly. Switching the order of environments additionally avoids learning effects. Consequently, we believe both the environments as well as the measurements described below evaluate CogniCrypt's effectiveness fairly and appropriately.

In summary, this study design leaves us with four different conditions. Condition 1 has the participant start with task FE using the regular Eclipse and then go on to implementing task TLS with CogniCrypt (FE/EC→TLS/CC). In condition 2, the order is swapped (TLS/CC→FE/EC). For condition 3, a participant first works on task TLS in regular Eclipse and subsequently continues with task FE in CogniCrypt (TLS/EC→FE/CC). Condition 4 once again switches the order of configurations from condition 3 (FE/CC→TLS/EC).

B. Participants and Experiment Context

We recruited 32 graduate students at two universities to participate in the experiment. All students were either currently taking a course including Java development tasks or had completed such a course already, e.g., a course for which they had implemented several static program analyses in Java. We considered this experience sufficient in terms of Java programming skills. We did not filter based on students' knowledge of Eclipse. During our recruitment, we did not mention cryptography to not bias our sample set towards students who feel more comfortable with cryptography. Participation in the study was voluntary and not required as part of a course.

C. Collected Measurements

To answer RQ_1 and RQ_2, we have compiled a score sheet of requirements that the implementation of each task needs to exhibit in order to count as *functionally correct* or *secure*, respectively. Table II shows the criteria for both tasks. The test cases that we enhanced each stub with also covered requirements for functional correctness that we were able to cover through a unit test case. In Table II we mark the requirements that correspond to a test case in a stub by '(test)'. We measure correctness and security of each participant by first running the test cases on their code and subsequently manually checking the compliance with the remaining functionality as well as the security criteria. The percentage of items covered are the *functionality* and *security score* of each task, respectively.

For an implementation of the FE task to be considered correct, a ciphertext file must exist that is different from the plaintext file, but not empty. The password provided through the stub must also be used to generate a cryptographic key. Finally, some form of encryption must be used—even if it is a self-implemented one—that encrypts the whole plaintext. Security-wise, the encryption configuration must be secure. That is, no insecure algorithms (e.g., DES) or block modes (e.g., ECB) must be used. The key must be derived securely from the password. That requires (1) the password to be used, (2) the key derivation to be conducted through PBEKeySpec, and (3) the PBEKeySpec to be used securely. In addition, the encryption must be prepared securely. That is, depending on the cipher mode the participant uses, they may need to provide an Initialization Vector (IV).

To implement task TLS correctly, the client must be able to send a message and receive the server's answer. When it sends the correct message, it should also handle the appropriate response from the server. The client must use the correct IP and port, set the correct key store, flush the write channel, and close the connection at the end. From a security perspective, we require the implementation to actually use TLS. The TLS connection must also be set up using an appropriate socket, e.g., through the Java Secure Socket Extension (JSSE). Lastly, the TLS connection must be configured to use secure cipher suites and only enable secure TLS protocols. A default configuration of a TLS connection set up through the JSSE allows both insecure cipher suites and TLS protocols. Participants therefore have to configure these themselves. Participants who cannot use CogniCrypt thus have to not only discover on their own that the default configuration is insecure, they also have to find out how to enable secure cipher suites and TLS protocols only.

To answer RQ_3, we also measure the time participants take to complete the task. We consider completion time as the time from when a participant starts to read the task description until they close the development environment. We intentionally include any time spent outside the IDE looking at online resources as we believe this is part of the time taken to complete the task. In other words, we do not pause the timer if the IDE loses focus.

D. Survey Questionnaire

To answer RQ_4 and RQ_5, we want to understand the steps developers take to solve a task. However, to ensure a natural work setting and to avoid inaccuracies in measuring completion time, we do not follow a "think aloud" approach [17]. Instead, we ask participants to fill out questionnaires after each task. In these questionnaires, we ask about the perceived difficulty of the task, the clarity of the task description, and their experience with the environment. The questions are available in Appendix A. We used Google Forms to create all three questionnaires.

E. Pre-Testing

We first conducted a pilot study with five test participants. For the purpose of the pilot study, we followed the same study design as described above, apart from one aspect: from participants of the pilot study we aimed at receiving feedback to refine our study design if necessary and were not attempting to take exact time measures. As a result, for the pilot study we *did* follow the "think aloud" approach to gather direct feedback from our pilot-study participants.

Participants informed us of ambiguities in the task descriptions and confusing oddities in some UI elements of CogniCrypt. For the final study, we revised the formulations in questions and re-designed the respective UI elements. All the pilot study participants finished both tasks, including questionnaires, within 45 to 60 minutes. In the final experiment, we hence told participants to finish within an hour. After half the time, we reminded them to move on to the second task if they had not already. From the results we report in this work, none have been gained from the pilot study.

TABLE III: Conditions & Participants

Condition	Particpants
EC/FE → CC/TLS	7
CC/TLS → EC/FE	4
EC/TLS → CC/FE	7
CC/FE → EC/TLS	6

IV. Results

From originally 32 participants, we had to exclude the results of eight because they neither executed CogniCrypt$_{GEN}$ nor CogniCrypt$_{SAST}$ throughout the whole study. We had not anticipated this scenario and had not set up any telemetry or questions in the evaluation survey. Thus, we have no further insights on why these eight participants refrained from using CogniCrypt. For the remaining 24 (*P01 – P24*) we show the participant distribution among the conditions in Table III. We manually analyzed the participants' code and their survey answers only for the remaining 24 participants. The manual analysis was conducted by the first and the second author. The agreement ratio for functionality and security score are 92% and 90%, respectively. For all differences in rating, the two raters negotiated until they reached a compromise. Table IV provides a complete overview of all results of the 24 participants.

A. Functionality (RQ_1)

For each solution, we first investigate whether it actually implemented the task completely. To this end, we first run the test cases. In the following, we will distinguish between running and broken solutions. For us to consider a solution *running*, all the provided unit test cases must terminate without exception, even if they fail. We consider a solution *broken*, on the other hand, when at least one of its test cases throws an exception. We make this distinction because non-running programs are distinctly non-functional in comparison to a program that does not implement all functionality, but at least terminates. To appropriately account for this, we award all non-running programs zero functionality points regardless of the state of their implementation and discard them from the remainder of this discussion.

Our results indicate that there is a noticeable difference between the running/broken ratios of solutions that have been implemented using CogniCrypt and those without. Without CogniCrypt, participants only produced running code for the FE task in six out of eleven cases. For task TLS, only two participants managed to get the test cases working. Two further participants succeeded at establishing a connection, but failed at sending data, causing the test cases to hang. Everyone else but two participants did not manage to establish a connection to begin with. In contrast, *with CogniCrypt*, participants produced running code for all but one case for both task FE (twelve out of thirteen) and task TLS (ten out of eleven).

Participant *P07* who did not manage to complete task FE with CogniCrypt did use CogniCrypt$_{GEN}$ and had it generate the method `templateUsage()` into the correct class. However, they then ignored the generated code and attempted to implement a custom solution that throws an exception when the test cases are run. Participant *P15* failed to implement task TLS for a similar reason. They also used CogniCrypt$_{GEN}$ to generate code, but for the encryption use case. They subsequently tried to manually set up the TLS connection and used the generated `templateUsage()` only to encrypt the message. As their code does not compile, the test cases cannot be executed.

As mentioned above, we will limit the following discussion to solutions that can be run. Figure 4 shows the functionality scores across all four combinations of environment/programming task. The score is shown in percentages, that is, a score of 2 out 4 is shown as 0.50. In our following discussion, when we justify scores, we refer to individual points instead of the percentages.

a) FE: For task FE, three of the six participants who completed the task *without* CogniCrypt achieve a full functionality score, resulting in a mean functional score of 1. We deducted one point from the other three solutions because they all failed to derive the encryption key from the password. From the twelve participants who implemented task FE successfully *with* CogniCrypt, eight did so with a full functionality score. Participant *P23* generated code for the wrong use case ("Secure Password Storage") and then attempted to use it in a custom-made encryption. The remaining three participants did manage to run CogniCrypt$_{GEN}$, but then failed to integrate the generated code in one way or another. *P02* completed the implementation of the encryption, but did not manage to store the result of the encryption in the variable the stub writes into the ciphertext file. Hence, the ciphertext file has the same content as the plaintext, although the encryption itself was implemented in a functionally correct manner. *P01* ignored the generated code. They even went so far as to delete the method `templateUsage()` from the Java file they coded in, but left the other generated code untouched. Instead, they implemented a custom solution that they did not manage to finish. Lastly, *P09*

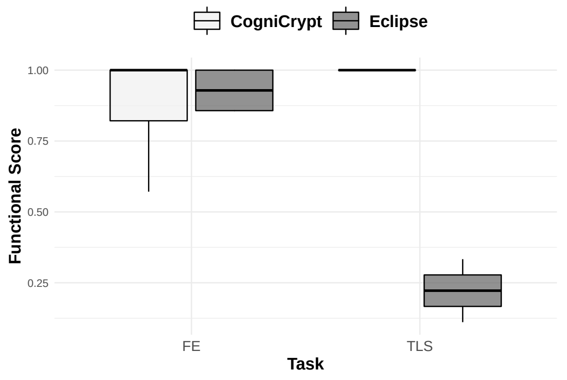


Fig. 4: Functionality Score

only made use of the key-generation code and attempted to implement a custom encryption solution for the data using `CipherOutputStream`. However, this solution does not encrypt the whole plaintext.

b) TLS: For task TLS, the results are much clearer than for task FE. First, as mentioned above, only two participants who did not use CogniCrypt for the task, did actually produce running code. Those two received two (*P23*) and three (*P21*) out of nine points on the functionality score, respectively. In neither submission do any of the test cases pass nor do they set the correct keystore. *P23*, in addition, fails to flush the channel to the server. In contrast to that, all ten participants who implemented task TLS using CogniCrypt received full points on the functionality score. They all generated code using CogniCrypt$_{GEN}$ and integrated it properly into the program stub.

c) Summary: In conclusion, for the eight participants who implemented task FE with CogniCrypt and received full points on the functionality score, CogniCrypt worked as intended. Some participants did, however, face problems. In one case, the participant was not clear about which use case they need to pick. For the remaining three, CogniCrypt failed at properly communicating that and how they need to integrate the generated code into their own application code. On the other hand, all but two participants attempting to implement task TLS without CogniCrypt failed to so, while all but one who did use CogniCrypt succeeded. For both tasks, we found participants achieved statistically significant better scores using a Wilcoxon signed-rank test for paired data ($p < 0.05$).

Further, we used the Wilcoxon signed-rank test to check for correlations of functionality score and self-reported experience in programming (Q15–Q17), Eclipse (Q18–Q20), security or cryptography (Q21–Q25) or general demographics (Q26–Q32), but were not able to find any. Similarly, we have not found any correlations between functionality score and order of task or tool.

B. Security (RQ_2)

We observe similar trends for the security score. We show the distribution over the four environment/task combinations

TABLE IV: Participants Overview

Participant	Condition		Score Task 1			Score Task 2		
	Task 1	Task 2	CogniCrypt	Functionality	Security	CogniCrypt	Functionality	Security
P01	TLS	FE	○	0/9	0/4	●	4/7	0/6
P02	FE	TLS	●	5/7	5/6	○	0/9	0/4
P03	FE	TLS	○	6/7	1/6	●	9/9	2/4
P04	TLS	FE	○	0/9	0/4	●	7/7	6/6
P05	FE	TLS	●	7/7	6/6	○	0/9	0/4
P06	FE	TLS	○	6/7	1/6	●	9/9	4/4
P07	TLS	FE	○	0/9	0/4	●	0/7	0/6
P08	FE	TLS	○	7/7	6/6	●	9/9	4/4
P09	TLS	FE	○	0/9	0/4	●	6/7	6/6
P10	FE	TLS	○	0/7	0/6	●	9/9	4/4
P11	TLS	FE	●	9/9	4/4	○	0/7	0/6
P12	FE	TLS	●	7/7	6/6	○	0/9	0/4
P13	FE	TLS	●	7/7	6/6	○	0/9	0/4
P14	FE	TLS	○	7/7	1/6	●	9/9	4/4
P15	TLS	FE	●	0/9	0/4	○	0/7	0/6
P16	TLS	FE	○	0/9	0/4	●	7/7	5/6
P17	TLS	FE	●	9/9	4/4	○	6/7	1/6
P18	TLS	FE	●	9/9	4/4	○	0/7	0/6
P19	FE	TLS	○	7/7	4/6	●	9/9	4/4
P20	TLS	FE	○	0/9	0/4	●	7/7	6/6
P21	FE	TLS	●	7/7	6/6	○	3/9	2/4
P22	FE	TLS	○	0/7	0/6	●	9/9	4/4
P23	TLS	FE	○	2/9	2/4	●	7/7	6/6
P24	FE	TLS	●	4/7	1/6	○	0/9	0/4

● indicates that the task was performed with CogniCrypt and ○ without, respectively.

in the box plot in Figure 5.

a) FE: For task FE, similar to the functional score, we find again somewhat ambiguous results, although much less so than for the functional score. Only one of the six participants who implemented task FE *without* CogniCrypt achieved a full security score. Participant *P19* achieved four out of six security points, only lacking a random salt, and choosing an insecure iteration count. The remaining four participants all received one out of six points. From the twelve participants who *did use* CogniCrypt, eight received a perfect score. We removed one point each for participants *P02* and *P16* because they transformed the password from a `char` array into a `String`. Neither of the two ended up using the String password variable, making it effectively dead code and likely to be optimized away by the Java compiler. We also assume this code would have been cleaned up in any real-world setting, but decided to remove the point nonetheless because the code as-is is insecure. *P01* and *P23*'s custom solutions, which we already discussed above, do not hold any security guarantees. While, for instance, *P01* generates a cryptographic key from the password (which is why they get a point on the functional score for this requirement), they do so using a number of String, hashing, and array-copy operations. The code also transforms the password into a `String`. In total, *P01*'s solution receives zero points.

b) TLS: Participants who implemented task TLS *with* CogniCrypt all received a perfect security score. This is because the code generated through CogniCrypt only enables secure cipher suites and TLS protocols. The two participants who implemented at least a *running* program for task TLS without CogniCrypt achieved zero (*P023*) and two points (*P021*), respectively. We removed two points for participant

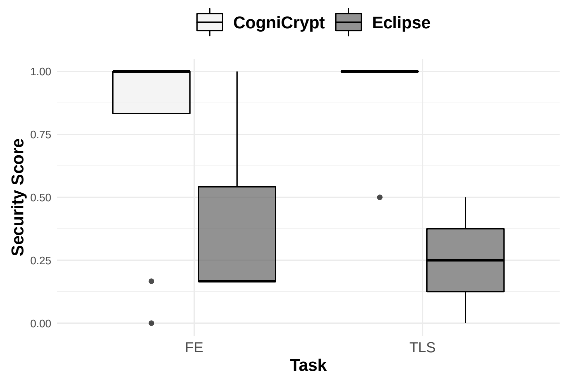


Fig. 5: Security Score.

P021 because they neglected to configure the connection in terms of cipher suites and TLS protocols.

We also checked the security of the broken solutions for task TLS developed without CogniCrypt to provide at least some kind of evaluation. None of the ten participants would receive more than two points because they display the same problem as *P21*'s solution. For the reasons we explained above, we do not include this data into the box plot, however.

c) Summary: In summary, participants fare better in terms of security for both tasks when using CogniCrypt, compared to when they try it without. As with the functionality score, we were able to show a statistically significant improvement with CogniCrypt, through a Wilcoxon signed-rank test for paired data ($p < 0.05$). For participants using CogniCrypt, the only points detracted were for failing to clean up the code and for complete custom-made solutions.

We have again checked for correlations with any of the forms of experience we surveyed participants about in the questionnaire as well as the order of tasks and tools using a Wilcoxon signed-rank test, but could not find any.

C. Completion Time (RQ_3)

We report the distribution of completion times in Figure 6. As with functional and security score, we only report completion times for non-broken solutions. We first note that completion times for participants using CogniCrypt spread comparatively widely. For task TLS, *P18* finished in six minutes and thirty-two seconds, whereas it took *P11* about 39 minutes and 30 seconds. The fastest successful participant for task FE completed their work in not even two minutes. In stark contrast, the participant who took the longest needed about 42 minutes. We attribute this wide range to two behaviours we observed while conducting the study when walking around the room and watching participants over the shoulder. Many participants, when they had CogniCrypt available, first attempted to finish the task without using either CogniCrypt$_{GEN}$ or CogniCrypt$_{SAST}$. Most eventually gave up, resorting to either launching CogniCrypt$_{GEN}$'s wizard or triggering CogniCrypt$_{SAST}$. Second, some participants took longer than others to generate code for the correct solution using CogniCrypt. Some appeared to struggle when having to answer questions in CogniCrypt$_{GEN}$'s wizard. Others even generated code for an incorrect use case at first.

a) FE: When comparing the two plots for completion time with task FE directly, participants are slightly faster with CogniCrypt, although there is no statistical significance using one-sided paired Wilcoxon test. The slowest participant *with* CogniCrypt is slower by several minutes than the slowest participant *without* CogniCrypt. The diagram presents a somewhat skewed picture, however, because more participants managed to finish when *using* CogniCrypt compared to when *not* using it (twelve of thirteen vs. six out of eleven). We find it likely that participants who took longer *with* CogniCrypt would not have finished if they had not had it at their disposal.

b) TLS: For task TLS, the median completion time lies at around fourteen minutes when using CogniCrypt. The two participants who finished TLS without CogniCrypt completed their work faster. However, as both the functional and security scores of the two indicate, their solutions are far from being actually complete and secure. On top of that, we also argue again that many participants who took longer with CogniCrypt would not have produced *running* code without it. The high number of participants who did not produce running code without CogniCrypt serves as a strong indicator for this claim. Given these two observations, we conclude that CogniCrypt improves the completion time for this task.

c) Summary: We conclude that participants are generally faster with CogniCrypt. We come to this conclusion because of (a) the higher completion rates in a setting with limited time available and (b) the lower median completion times for the solutions that were completed. Where CogniCrypt seems to be slower than regular Eclipse, we argue the slower speed to be more indicative of more people being enabled to finish a task to begin with.

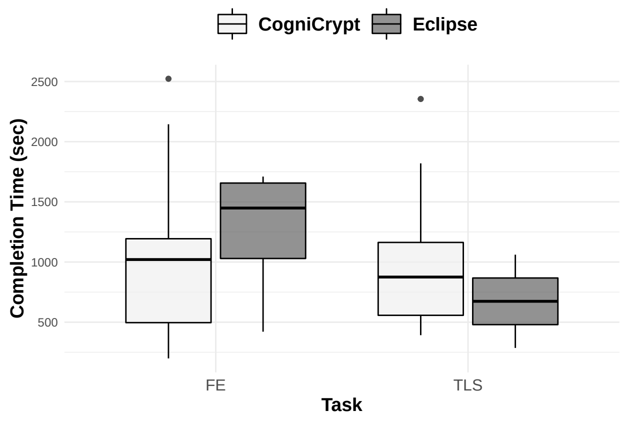


Fig. 6: Completion Time.

D. Usability (RQ_4)

Participants generally expressed positive views on CogniCrypt's usability, but provided criticism relating to the integration with Eclipse of both CogniCrypt$_{GEN}$ and CogniCrypt$_{SAST}$. In contrast, regular Eclipse overall received substantially worse reviews by participants. These results are reflected in the both tools' NPS values. NPS generally measures user satisfaction and ranges from -100 to 100, whereas any value above 50 is considered excellent and values below 0 are considered bad [27]. CogniCrypt receives an NPS value of 33.33, a result that is generally considered good, but not excellent [27]. Regular Eclipse, on the other hand, receives an NPS of -54.17.

When asked for more concrete feedback (Q7/Q14), participants especially praised CogniCrypt$_{GEN}$ (*P02* – *P07*, *P09*, *P11*, *P12*, *P16*, *P19*, *P21*). *P02* notes that they '*have never worked on encryption before in Java.*' However, they would '*strongly say that [CogniCrypt] will be very helpful with prior knowledge [of the tool].*' *P05* adds they found CogniCrypt$_{GEN}$ useful and that it was '*easy to choose input and generate the encryption function by the tool. It was very user-friendly for developer since there was no further need to read about the encryption function and security/authenticity of that encryption library*'. Both aspects highlighted by *P05*—the high effectiveness for developers with low experience and that CogniCrypt lifts a burden off a developer because it takes care of security features—are also shared by other participants. *P19*, for instance, finds CogniCrypt$_{GEN}$ 'very helpful' because they could '*assume[...] that [the] generated code [was] secure, which saved a lot of time searching for documentation on security requirements*'. *P16* also highlights the usefulness for cryptography novices: '*Since I have never worked on anything similar before, the code generator ([for task TLS]) was very helpful for me to get an idea about what I* *have to do. It was easy to use, since every single configuration was asked.*' *P12* agrees as they '*haven't worked that much with encryption so far, which is why the code generation was especially helpful.*' *P04* highlights the effort that is saved through CogniCrypt$_{GEN}$, because they do not '*have to search the web for the right and required classes to get the job done.*' *P03* goes so far as considering code generators like CogniCrypt$_{GEN}$ necessary for '*secure software engineering*', because '*we can be sure our code is almost secure.*'

P06 appreciates CogniCrypt$_{SAST}$ as helpful because it detected their use of an inappropriate block cipher mode for the call `Cipher.getInstance('AES')`. *P13* tested CogniCrypt$_{SAST}$ by purposefully introducing misuses to the code and enjoyed that it found them. *P23* praised CogniCrypt to be '*well integrated*' and *P20* found it generally '*easier to use*'.

a) Summary: CogniCrypt has fared significantly better than Eclipse. Participants generally praise the ease of use and the help they receive, in particular from CogniCrypt$_{GEN}$. Furthermore, CogniCrypt's NPS result not only trumps Eclipse's NPS value by a large margin, but is considered good. However, it also shows that the tool still comes with limitations, which we further discuss in the next section.

E. Obstacles (RQ_5)

One obstacle several participants clearly faced was the integration of generated code into their project. This obstacle is, on the one hand, demonstrated by the three participants we described above who had CogniCrypt$_{GEN}$ generate code for task FE, but then failed to properly call that code. However, participants have also mentioned CogniCrypt's shortcomings in that respect in their feedback. *P21* comments that '*it was difficult to find where the autmomatic generated code is located*'. *P01* requests the '*comments should be improved for auto-generated code*', because it took them '*a while to understand what the generated code says and to identify which part of the code is generated when [they] already have [their] own code in the class. So there should be clear way to separate automatic generated code from [their] own code.*' *P08* encountered the same problem as they needed time to understand which classes CogniCrypt$_{GEN}$ had generated for them to implement task TLS. When implementing task FE, *P13* struggled to identify, if the generated code would already derive the key from the password or if they had to implement that themselves. In the end, they did not implement it themselves, because the tests passed.

One further problem with CogniCrypt$_{GEN}$ in particular appears to be the usability of its wizard. As we noted above, several participants had to go through multiple attempts of generating code. The issue is also highlighted by *P22* who criticises that they at first '*didn't notice the first screen of the wizard provided a choice, [they instead] mistook it for an introductory page and just clicked 'continue'[, and were] then confused that the code generated didn't suit my needs*'.

The warning messages by CogniCrypt$_{SAST}$ provided another major obstacle participants reported on. *P13* rightly complains that it '*was confusing [...] that the autogenerated code throws errors (in [CogniCrypt$_{SAST}$]), which on further inspection are only in the decryption case and not applicable.*' *P02* ran into the same issue when they were '*getting a[n] error at [the encryption] method at line number 67. Which I could not understand that what is exactly the problem.*' The warnings the two participants report of relate to long-known false positives in CogniCrypt$_{SAST}$ that have been addressed since conducting the study. *P07* raises a further issue with CogniCrypt$_{SAST}$'s reporting: '*But when we are using functions with more than two arguments [CogniCrypt$_{SAST}$ only says] the parameter is not properly generated, additionally it should also return info about how to correct it or some possible description for the developer to enhance the code*'. The warnings, both *P02* and *P07* refer to, CogniCrypt$_{SAST}$ displays for predicate violations. The developers of CogniCrypt have indeed been struggling to find a good wording for these kinds of violations and have been editing CogniCrypt$_{SAST}$ warning messages for such misuses upon feedback by users several times since CogniCrypt$_{SAST}$'s original publication [14]. This comment shows there is still work to be done regarding this matter.

Lastly, *P01* criticises that CogniCrypt$_{GEN}$ and CogniCrypt$_{SAST}$ '*lack documentations, tooltips which detail the options they provide.*' This point is valid for the study, its applicability in practice, however, may be limited. As mentioned before, we intentionally kept the documentation limited to not unnecessarily bias participants. In real-world contexts, users would have the documentation available on CogniCrypt's website[1], which provides extensive introductions to all its components.

a) Summary: We conclude that while CogniCrypt has generally received favourable feedback, it still exhibits severe usability shortcomings. Some participants have struggled with integrating the code generated by CogniCrypt$_{GEN}$, but even more face problems understanding and interpreting error messages by CogniCrypt$_{SAST}$.

V. Discussion

Participants in our controlled experiment were significantly faster in implementing application code that requires using cryptography concepts. The code they produced was significantly more functional *and* secure than when only using Eclipse. Participants generally judge CogniCrypt to be a useful and usable tool as demonstrated by its NPS score. Our results therefore allow us to conclude for RQ_1 to RQ_3 that CogniCrypt has a significant positive impact on all three. In addition, we can answer RQ_4 such that developers do indeed seem to view CogniCrypt as more usable than plain Eclipse. Our study, in conclusion, provides strong evidence that CogniCrypt is effective at combatting cryptographic misuse.

However, there is still room for improvement. In response to RQ_5, we can report two main findings. First, when it comes to integrating code by CogniCrypt$_{GEN}$ into their

[1]www.cognicrypt.org

application, a large subset of participants struggled because they had trouble understanding what is happening in their IDE when CogniCrypt$_{GEN}$ generated code. To help with the situation, CogniCrypt$_{GEN}$ should inform users better. CogniCrypt$_{GEN}$ should enhance method `templateUsage()` with a comment describing which pieces of code have been generated, what their purposes are, and where they each can be found. A second issue raised by participants revolves around the error messages produced by CogniCrypt$_{SAST}$, in particular error messages on predicate-related misuses. Participants find the warning messages too abstract and that they expect too much knowledge of CrySL for the average developer. They also criticise that they are not actionable, that is, they do not provide help as to how to resolve them. One possible avenue of future work might explore more focused usability testing for CogniCrypt$_{SAST}$'s error messages to A/B test alternative phrasings. One might even stop showing transitive predicate warnings entirely.

VI. Threats to Validity

Our sample set poses an internal threat to the experiment's validity because it consisted of only 32 graduate students from two universities. That sample set is likely not representative of the whole Java developer community. However, we argue that students are more likely to be less experienced in programming, especially programming in a specific language, than the average developer. We further assume grad students are only, if at all, slightly unrepresentative in terms of security knowledge. If CogniCrypt manages to support the demographic of students effectively, we expect it to fare even better with professional developers as they will likely have more experience with IDE-integrated code generators and program analysers. Recent work by Naiakshina et al. [23] supports this assumption empirically.

Our experiment further exhibits an ecological threat as well. Although we tried to come up with seemingly realistic tasks, both the task descriptions and the stubs are fairly unlikely to be found in practice in exactly this manner. The former rather resemble assignments in a programming course, the latter are comparatively tiny. It is further questionable as to how far the regular-Eclipse condition can be claimed to simulate an authentic environment for everyday development. Developers often configure their editor and IDEs to their liking. They also use a wide range of editors and IDEs to develop in Java and some participants had no experience in Eclipse. To mitigate this threat, we asked the participants for their experience level with Eclipse and found no statistically significant correlation with their scores or completion rates in the study. On top of that, these restrictions apply to lab studies in general and our study does not display stronger limitations than other comparative ones. As a result, we do not believe the setting albeit not necessarily representative of developer's work environment to cause the study's results to be of less significance.

VII. Related Work

In this section, we discuss tools similar to CogniCrypt and how they are evaluated. We also relate our study to prior work on empirical work about software security techniques.

A. Assistance for Cryptographic APIs

Apart from CogniCrypt, multiple tools have previously attempted to address the misuse of cryptographic [4, 5, 24, 28] and other security [8, 10, 30] APIs. However, these tools are limited to this misuse detection. On the other hand, tools such as CDRep [18] and FireSecBugs take a more directly corrective approach by repairing faulty programs.

In addition to the analysis these tools provide, CogniCrypt aims at supporting developers additionally through a use-case-based code generation that lifts the burden of handling low-level cryptographic APIs from the developers. In addition, CogniCrypt allows the coverage of its analysis to be expanded by means of API-usage specifications in an external specification language.

Kane et al. [12] suggest high-level abstractions on top of low-level cryptographic APIs. Their higher abstraction levels automatically avoid common misuses (e.g., hard-coded keys or initialization vectors). The client programs using those abstractions end up significantly smaller than semantically comparative programs that use traditional libraries. The wrapper code generated by CogniCrypt is similar as it lifts the abstraction level that the developer interacts with. Unlike CogniCrypt, those high-level abstractions are not use-case-based because they still require developers to know which cryptographic concepts they should use in a given use case. Consequently, developers may still misuse them, and are not helped by a misuse detector.

Apart from CogniCrypt by Krüger et al. [13] and Fixdroid by Nguyen et al. [24], none of the tools discussed above come with any form of IDE integration. Additionally, only Fixdroid has been previously evaluated in terms of usability and usefulness for developers. The other tools have solely been evaluated in terms of their capabilities to find misuses in existing applications. In light of this existing body of research on fixing cryptographic-API misuse, we identify an empirical gap with regard to the effectiveness of combining code generation and misuse detection. As a result, we have selected CogniCrypt as our evaluation object for this study to work towards closing this gap.

B. Evaluation of Other Software Security Techniques

Our experiments expand on previous empirical research on the effectiveness on counter measures against software insecurity. Prior work has investigated several aspects relating to software security: the role of a security-focused development process [2, 29], the content, length, and desired structure of security warnings [9], the role of resources for security knowledge [1, 6], as well as the effect of explicitly requesting developers towards writing secure code (i.e., priming) on its security [20, 21, 22].

Our work most closely resembles, in design and goal, that of Nguyen et al. [24]. The main difference lies in the object of evaluation. CogniCrypt combines a static misuse detector with a code generation, supports Java, and is integrated into Eclipse, while Fixdroid is only equipped with an analysis, is limited to Android, and integrates with Android Studio. Fixdroid further only checks for a few hard-coded misuses, whereas CogniCrypt's analysis component may be parameterized by usage specifications in CrySL. However, in contrast to CogniCrypt, Fixdroid supports IDE-integrated fixes that are selectable by users.

VIII. Conclusion

Cryptography can help secure sensitive data, but this help often ends up ineffective because cryptography is not integrate securely into applications. Krüger et al. [13] have proposed the cryptographic assistant CogniCrypt to tackle this problem and support application developers write secure code. The authors, however, did not empirically validate their claims. In this paper, we have filled this gap by conducting a controlled experiment investigating CogniCrypt's capabilities in reducing cryptographic misuse. Through this experiment, we have demonstrated CogniCrypt's effectiveness. The tool significantly improves how fast, functional, and secure developers code cryptography applications. Participants' criticism where they mentioned it related mostly to implementation details that can and should be addressed, but that also in no way threaten the validity of the concept underlying CogniCrypt— or even its concrete prototypical implementation.

References

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. How internet resources might be helping you develop faster but less securely. *IEEE Security & Privacy*, 15(2):50–60, 2017.

[2] Hala Assal and Sonia Chiasson. Security in the software development lifecycle. In *Fourteenth Symposium on Usable Privacy and Security, SOUPS 2018, Baltimore, MD, USA, August 12-14, 2018.*, pages 281–296, 2018.

[3] Gary Charness, Uri Gneezy, and Michael A Kuhn. Experimental methods: Between-subject and within-subject design. *Journal of Economic Behavior & Organization*, 81(1):1–8, 2012.

[4] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. Evaluation of cryptography usage in android applications. In *International Conference on Bio-inspired Information and Communications Technologies*, pages 83–90, 2016.

[5] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *ACM Conference on Computer and Communications Security*, pages 73–84, 2013.

[6] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 121–136, 2017.

[7] Lei Gao. Latin squares in experimental design. 2005.

[8] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Conference on Computer and Communications Security (CCS)*, pages 38–49, 2012.

[9] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl. Developers deserve security warnings, too: On the effect of integrated security advice on cryptographic API misuse. In *Fourteenth Symposium on Usable Privacy and Security, SOUPS 2018, Baltimore, MD, USA, August 12-14, 2018.*, pages 265–281, 2018.

[10] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, V. N. Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. Vetting SSL usage in applications with SSLINT. In *IEEE Symposium on Security and Privacy*, pages 519–534, 2015.

[11] Natalia Juristo and Ana M Moreno. *Basics of software engineering experimentation.* Springer Science & Business Media, 2013.

[12] Christopher Kane, Bo Lin, Saksham Chand, and Yanhong A. Liu. High-level cryptographic abstractions. *CoRR*, abs/1810.09065, 2018. URL http://arxiv.org/abs/1810.09065.

[13] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. Cognicrypt: supporting developers in using cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 931–936, 2017.

[14] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *European Conference on Object-Oriented Programming (ECOOP)*, 2018.

[15] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *IEEE Transactions on Software Engineering (TSE)*, 2019.

[16] Stefan Krüger, Karim Ali, and Eric Bodden. CogniCrypt$_{GEN}$ - Generating Code for the Secure Usage of Crypto APIs. In *Internationl Symposium on Code Generation and Optimization (CGO)*, 2020.

[17] Clayton Lewis. *Using the" thinking-aloud" method in cognitive interface design.* IBM TJ Watson Research Center Yorktown Heights, NY, 1982.

[18] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. Cdrep:

Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 711–722, 2016.

[19] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: why do Java developers struggle with cryptography APIs? In *International Conference on Software Engineering (ICSE)*, pages 935–946, 2016.

[20] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. Why do developers get password storage wrong?: A qualitative usability study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 311–328, 2017.

[21] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith. Deception task design in developer password studies: Exploring a student sample. In *Fourteenth Symposium on Usable Privacy and Security, SOUPS 2018, Baltimore, MD, USA, August 12-14, 2018.*, pages 297–313, 2018.

[22] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, Emanuel von Zezschwitz, and Matthew Smith. "if you want, i can store the encrypted password." - a password-storage field study with freelance developers. 2019.

[23] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, and Matthew Smith. On Conducting Security Developer Studies with CS Students: Examining a Password-Storage Study with CS Students, Freelancers, and Company Developers. In *CHI*. ACM, 2020. To appear.

[24] Duc-Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A stitch in time: Supporting android developers in writingsecure code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1065–1077, 2017.

[25] Oracle. Java cryptography architecture (JCA), 2017. https://docs.oracle.com/javase/9/security/java-cryptography-architecture-jca-reference-guide.htm#GUID-2BCFDD85-D533-4E6C-8CE9-29990DEB0190.

[26] Oracle. Java secure socket extension (JSSE), 2017. https://docs.oracle.com/javase/9/security/java-secure-socket-extension-jsse-reference-guide.htm#JSSEC-GUID-93DEEE16-0B70-40E5-BBE7-55C3FD432345.

[27] Frederick F Reichheld. The one number you need to grow. *Harvard Business Review*, 81(12):46–55, 2003.

[28] Shuai Shao, Guowei Dong, Tao Guo, Tianchang Yang, and Chenjie Shi. Modelling analysis and auto-detection of cryptographic misuse in Android applications. In *nternational Conference on Dependable, Autonomic and Secure Computing*, pages 75–80, 2014.

[29] Tyler W. Thomas, Madiha Tabassum, Bill Chu, and Heather Lipford. Security during application development: an application security expert perspective. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*, page 262, 2018.

[30] Qing Wang, Juanru Li, Yuanyuan Zhang, Hui Wang, Yikun Hu, Bodong Li, and Dawu Gu. Nativespeaker: Identifying crypto misuses in android native code libraries. In *Information Security and Cryptology - 13th International Conference, Inscrypt 2017, Xi'an, China, November 3-5, 2017, Revised Selected Papers*, pages 301–320, 2017.

[31] Eclipse Website. http://www.eclipse.org, 2020.

Appendix

Questionnaire

For each task, the questionnaire includes the following questions:

Q1/8: This task was difficult.
– Type: Scale 1 (Strongly Disagree) to 5 (Strongly Agree).

Q2/9: This task was fun.
– Type: Scale 1 (Strongly Disagree) to 5 (Strongly Agree).

Q3/10: I think I solved this task correctly.
– Type: Scale 1 (Strongly Disagree) to 5 (Strongly Agree).

Q4/11: I think I solved this task securely.
– Type: Scale 1 (Strongly Disagree) to 5 (Strongly Agree).

Q5/12: Have you written or seen code for tasks similar to this one before? For example, maybe you worked on a project that included a similar task, but someone else wrote that portion code.
– Type: Single-answer question.
– Possible choices: [I have written similar code; I have seen similar code, but have not written it myself; No, neither; I don't know; I prefer not to say.]

Q6/13: Based on the last task, please how likely you would recommend the coding environment to a fellow programmer.
– Type: Scale 0 to 10.

Q7/14: Please provide additional feedback about what you found useful or not useful in the coding environment. Please provide concrete examples when possible.
– Type: Free-text field.

We also provide another questionnaire at the end of the experiment (i.e., after finishing both tasks) that asks participants about their programming experience **Q15** –**Q20**, security expertise **Q21** – **Q26**, general demographics **Q27** – **Q31**, and more general comments on their experience **Q32**. This final questionnaire includes the following questions:

Q15: How many years have you been programming in Java?
– Type: Free-text field.

Q16: How many years have you been coding in general?
– Type: Free-text field.

Q17: How did you learn to code?
– Type: Multiple-answers question.

– Possible choices: [Self-taught, Online class, College, On-the-job training, coding, other]

Q18: Which coding environment do you primarily use (name of the IDE or text editor)?

– Type: Free-text field.

Q19: Are you currently using the Eclipse IDE?

– Type: Single-answers question.
– Possible choices: [Yes, No]

Q20: How long have you been using the Eclipse IDE (in years)?

– Type: Free-text field.

Q21: Do you have an IT-security background?

– Type: Single-answers question.
– Possible choices: [Yes, No, Prefer not to say]

Q22: If you answered yes in the previous question, please specify.

– Type: Free-text field.

Q23: Which of the following options do describe your experience with cryptography best?

– Type: Single-answers question.
– Possible choices: [I have never used cryptography during software devlopment, I invented my own cryptographic algorithm or protocol, I implemented a cryptographic algorithm or protocol, I occasionally use cryptographic APIs or libraries during software development, I don't know, Prefer not to say]

Q24: Have you taken a computer-security class or course in the last five years?

– Type: Single-answers question.
– Possible choices: [Yes, No, Prefer not to say]

Q25: If you answerd yes in the previous question, please specify.

– Type: Free-text field.

Q26: Please tell us your highest degree of education.

– Single-answer question.
– Possible choices: [Less than high school, High school, Some college, Bachelor's degree, Master's degree, Professional degree, Ph.D., Prefer not to say.]

Q27: Please tell us your gender. ("na" for "prefer not to say")

– Type: Free-text field.

Q28: How old are you?

– Type: Free-text field.

Q29: What country to you live in?

– Type: Free-text field.

Q30: What is your native language?

– Type: Free-text field.

Q31: What other languages are you fluent in (if any)?

– Type: Free-text field.

Q32: Please provide any additional feedback for the study you have.

– Type: Free-text field.