Less is More: Supporting Developers in Vulnerability Detection during Code Review

Larissa Braz
larissa@ifi.uzh.ch
University of Zurich

Christian Aeberhard
christian.aeberhard2@uzh.ch
University of Zurich

Gül Çalikli
handangul.calikli@glasgow.ac.uk
University of Glasgow

Alberto Bacchelli
bacchelli@ifi.uzh.ch
University of Zurich

ABSTRACT

Reviewing source code from a security perspective has proven to be a difficult task. Indeed, previous research has shown that developers often miss even popular and easy-to-detect vulnerabilities during code review. Initial evidence suggests that a significant cause may lie in the reviewers' mental attitude and common practices.

In this study, we investigate whether and how explicitly asking developers to focus on security during a code review affects the detection of vulnerabilities. Furthermore, we evaluate the effect of providing a security checklist to guide the security review. To this aim, we conduct an online experiment with 150 participants, of which 71% report to have three or more years of professional development experience. Our results show that simply asking reviewers to focus on security during the code review increases eight times the probability of vulnerability detection. The presence of a security checklist does not significantly improve the outcome further, even when the checklist is tailored to the change under review and the existing vulnerabilities in the change. These results provide evidence supporting the mental attitude hypothesis and call for further work on security checklists' effectiveness and design.

Preprint: https://arxiv.org/abs/2202.04586

Data and materials: https://doi.org/10.5281/zenodo.6026291

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Software evolution.*

KEYWORDS

code review, security vulnerability, checklist, mental attitude

ACM Reference Format:
Larissa Braz, Christian Aeberhard, Gül Çalikli, and Alberto Bacchelli. 2022. Less is More: Supporting Developers in Vulnerability Detection during Code Review. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3511560

1 INTRODUCTION

A *vulnerability* is a "flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy" [71]. The later vulnerabilities are discovered in the software development cycle, the higher the associated fixing costs are [56]. Therefore, to avoid vulnerabilities, organizations are shifting security to earlier stages of software development [2]. However, security experts have to motivate and convince developers of the importance of finding vulnerabilities [80]. Yet, where to locate security within an organization remains a challenge [79]. For instance, a programmer working solo is likely to create avoidable security problems because they can naturally have only one point of view [86]. A solution to avoid these issues can be adopting security practices during code review.

Code review is a widely agreed-on practice [15] recognized as a valuable tool for reducing software defects and improving the quality of software projects [3, 4, 10]. Previous studies show that code review is also an important practice for detecting and fixing security bugs earlier [46, 81] and has positive effects on secure software development [48, 49, 70]. However, adopting security practices requires a large amount of knowledge which takes time to learn, and it can be hard to motivate [57, 83]. In fact, security issues (even popular ones, such as *Sensitive Data Exposure* [65]) still often reach production code, despite code review practices. So, *how can we better support code reviewers in detecting vulnerabilities?*

In the study we present in this paper, we investigate three interventions that aim to tackle the problem by guiding the focus of the reviewer. One is based on the *developer's mental attitude hypothesis* and two are based on an additional *security checklists hypothesis.*

Studies in the literature indicate developers' mental attitude as a leading cause for the introduction of vulnerabilities in the code [50, 88, 91]. Specifically, vulnerabilities may be introduced because developers do not consider security as their responsibility [50] or strongly rely on other project members, processes, and technologies [88]. A potential solution to resolve issues related to developers' mental attitude is giving explicit instructions regarding security. Indeed, Naiakshina et al. [51, 52] showed positive effects when explicitly instructing computer science students and freelance developers to implement *secure* password storage during coding. Nevertheless, writing and reviewing code are different activities [41], with even cultural differences among teams [10], therefore evidence about the former may not translate to the latter. Some preliminary yet promising evidence exists that the developers' mental attitude could play a role in the context of code review too [17]. Using a one-group pretest-posttest experimental design [24], Braz et al. [17] found that a significant number of developers who missed a popular vulnerability during a code review could find it when explicitly warned about its presence.

In the context of code inspections [30], the use of checklists to support developers has been extensively studied [44, 54, 58, 68] with positive results [26]. The OWASP foundation [33] proposes a popular code review guide that contains a security checklist [63]. It comprises items that guide the developers in finding security issues

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3511560

during a review. Despite the positive results of using checklists during code inspection and the efforts by the OWASP foundation, the effectiveness of checklists in contemporary code review practices and to support vulnerability detection has not been established yet.

In our study, we set up three interventions (treatments) that we compare among themselves and to a review baseline (control). The baseline (No security Instructions–**NI**) consists of asking developers to perform a code review without giving special instructions. The first treatment (Security Instructions–**SI**), based on the mental attitude hypothesis, explicitly instructs developers to perform the review from a security perspective. The second (Security Checklist–**SC**) and third (Tailored security Checklist–**TC**) treatments additionally ask developers to use a security checklist in their review. The **SC** checklist is derived from OWASP's Code Review Checklist [63], while the **TC** one is a shorter version tailored to the change and security issues at hand, which we created to remove confounding factors caused by the length of the normal checklist.

We implement our study as an online experiment. A total of 150 valid participants completed it. Among our participants, 62% (93) reported being software engineers, 71% (106) reported three years or more of professional development experience, and 65% (97) reported performing code reviews daily.

Our results support the mental attitude hypothesis: Participants instructed to focus on security issues were eight times more likely to detect vulnerabilities. Our results also call for further work on security checklists' effectiveness and design as they do not increase the detection of vulnerabilities in a security-focused review.

2 BACKGROUND AND RELATED WORK

In this section, we briefly introduce the concepts of code inspections and modern code review (the context of our study). Then, we review the literature on explicitly asking developers to use secure coding practices as a way to overcome issues due to the mental attitude and on checklists for secure software development and inspections/reviews. We also provide background on the two security vulnerabilities we focus on in our online experiment.

Software developers and security. Security often fails because of the lack of usability: users either misunderstand the security implications of their actions or turn off security features to workaround usability problems [11]. Software developers need to design systems that are both usable and secure. Yet, they still need support to create secure applications before addressing usable security [79].

Existing findings about developers' attitude towards security are not conclusive. On the one hand, a number of studies [7, 57, 90] found that developers prioritize more-visible functional requirements or even easy-to-measure activities, such as closing bug tracking tickets, over security. On the other hand, Christakis and Bird [21] reported that developers care more about security issues than other reliability issues. Smith et al. [75] advocate that static analysis tools detect vulnerabilities and help developers resolve those vulnerabilities. Previous studies have proposed and improved tooling support according to developers' needs [8, 9, 74], but tools are still generally poorly adopted by developers [79] as they are confusing for developers to use [75].

Developer-Centred Security (DCS). DCS studies have addressed some of developers' needs and attempted to apply existing methodologies from Human Computer Interaction and to adopt well-established usable security measures to software development [39, 55, 89]. The systematic literature review by Tahaei and Vaniea [79] points out the lack of research on several aspects of **DCS**. For instance, they reported only one study in the context of code review [28].

Code inspections and modern code review. Peer code review is a manual inspection of source code by developers other than the author. In 1976, Fagan [29] formalized a highly structured process for code reviewing, which includes a synchronous inspection meeting–code inspections. Over the years, researchers conducted several empirical studies on code inspections [42].

Nowadays, most organizations adopt a more lightweight code review process to limit the inefficiencies of inspections: modern code review [22]. This form of review is the focus of our study. Modern code review is asynchronous, tool- and change-based [12], and widely used in practice nowadays across companies [10, 69] and community-driven projects [66, 67].

Contemporary code review mitigates several issues of code inspections but also removes the structure and checks devised to keep the reliability of the process and its outcome. Researchers provided strong evidence that, as a result of this different approach to inspection, the outcome of contemporary code review is less predictable and past theories from code inspections do not transfer seamlessly [10, 12, 45]. Therefore, it is important to conduct studies specific to this different context.

Explicitly asking for secure coding practices. Naiakshina et al. [51, 52] conducted two studies with 40 computer science students. Half the participants received a task description that did not mention security, while the other half were explicitly tasked with implementing a secure solution. None of the participants in the first group stored passwords securely; while twelve participants in the group asked to create a secure solution implemented some level of security. Moreover, Naiakshina et al. [50] performed a similar study with freelance developers and found significant positive effect of security prompting on the secure storage of passwords.

Weir et al. [85] interviewed twelve industry experts to investigate how to improve the security skills of mobile app developers. They found that many of the most effective techniques for finding security issues are *dialectic*, *i.e.,* the discovery of knowledge through one person questioning another. Some of the techniques recommended were penetration testing, code review, pair programming, and a variety of code analysis tools. On the contrary, Edmundson et al. [28] stated that manual code review could be expensive and impractical due to the need for several reviewers to inspect at a piece of code to find a vulnerability. Later, Braz et al. [17] conducted a one-group pretest-posttest code review experiment with software developers. They found that several developers often miss a popular and easy-to-detect vulnerability when reviewing code (pretest); yet, when explicitly informed about the presence of a vulnerability in the change, a significant portion of the additional developers could identify it (posttest). These studies provide initial evidence that a specific, focused instruction on security could steer the developers' attention, thus overcoming their common mental attitude to not consider security aspects in review.

Checklist-supported inspections and review. Checklists have been mostly investigated for code inspections. Ad Hoc Reading

(AHR) and Checklist-Based Reading (CBR) are the standard reading techniques adopted by during code inspections [43]. In a study with undergraduate students, Oladele and Adedayo [54] found that CBR is effective in finding more issues during inspections with a 50% decrease in false positives compared to AHR. On the other hand, Akinola and Osofisan [5] did not find any significant differences in the efficiency of AHR and CBR in their study conducted with students in a distributed environment. Similarly, the findings of Lanubile and Visaggio [44] and Porter et al. [58] showed no significant differences between AHR and CBR in terms of the number of defects detected during software requirements inspections.

Studies on checklists for contemporary code review are fewer. Rong et al. [68] conducted a semi-controlled experiment with students and found evidence that checklists can help in guiding them during code reviews. In the context of education, Chong et al. [20] found that students are able to anticipate potential defects and create a relatively good code review checklist, which can be used to find defects. Finally, Gonçalves et al. [38] registered an experiment report with the goal of investigating whether review checklists and guidance improve code review performance.

Checklists in software security. Previous studies have addressed the design of checklists for security-related aspects, such as software security requirements and security life-cycle [6, 34, 37]. Gilliam et al. [37] provided guidelines for creating a software security checklist, emphasizing the importance of verifying security requirements beforehand to create a viable checklist. Garrison and Posey [34] suggested that security checklists are particularly useful in guiding non-security professionals through a security-oriented software development process. OWASP [33] stated that organizations with a proper code review process integrated into the software development life-cycle produced remarkably better code from a security standpoint [63]. To help businesses strengthen their security, such organizations developed different secure coding practices, which are widely adopted globally [73]. In fact, they proposed a code review guide containing a security checklist [63].

Cruzes et al. [25] investigated secure coding checklists' usage. However, the authors disagree that developers should adopt this practice. Smarjov [73] assessed the OWASP checklist's effectiveness on the number of vulnerabilities reported to the HackerOne platform [40]. They found a moderate connection between filling out the checklist during the development phase and the distribution of the vulnerabilities reported in HackerOne. The likelihood of HackerOne participants finding a new vulnerability is 2.92 times higher if they do not follow the checklist during the code development. To best of our knowledge, no study has been conducted to investigate the impact of security checklists for security-focused code review. We address this question by analyzing whether a security checklist supports developers in this task.

Software Vulnerabilities. OWASP [33] and the Common Weakness Enumeration Project (CWE) [1] are well-established projects that provide guidance and best practices for organizations to avoid security issues. Our study investigates whether security checklists support developers in vulnerability detection during code reviews. Therefore, among the OWASP's Top 10 Web Application Security Risks list [64], we focus on the *Sensitive Data Exposure* (*SDE*– CWE 1029) [59] group, which can be mapped to items in our checklist. *SDE* occurrences have increased significantly over the last few years [72], leading to damaging leaks. For instance, the largest instance of *SDE* to date impacted 3 billion Yahoo!'s user accounts, leaking their credentials (*i.e.,* email addresses, passwords, and security questions and answers). Security experts noted that the majority of the passwords used a strong hashing algorithm, but many used the *MD*5 algorithm, which can be broken rather quickly [87].

There are many reasons for *SDE*, such as the Missing Encryption of Sensitive Data (CWE-311) [61] and the ones covered in our experiments, namely *Generation of Error Message Containing Sensitive Information* (**MSI**- CWE 209) [60] and *Use of a Broken or Risky Cryptographic Algorithm* (**BRA**- CWE 327) [62]. The former refers to software that generates an error message including sensitive information about its environment, users, or associated data; while the latter refers to the use of a not recommended algorithm that may allow attackers to compromise data that has been protected.

3 RESEARCH METHODOLOGY

In this section, we introduce our research questions and detail our experimental design.

3.1 Research Questions

We structure our study around two research questions. We aim to understand the impact of (1) instructing developers to focus on security issues and (2) providing an additional checklists on vulnerability detection.

We formulate our first research question as follows:

> **RQ$_1$.** *Does explicitly asking developers to focus on security issues facilitate vulnerability detection during code review?*

We hypothesize that explicitly asking developers to focus on security issues increases vulnerability detection during in code review, because it would steer developers' mental attitude. Our formal hypothesis for **RQ$_1$** is:

H0$_1$: The presence of security instructions does not facilitate vulnerability detection during code review.

Evidence shows that generic checklists aid developers during code inspections and can improve the inspections' outcome [54, 68]. We explore the hypothesis that providing developers with a security checklist (in addition to instructing them to do a security review) further increases the vulnerability detection during code review because it would guide the review tasks on relevant security aspects. We formulate our second research question as follows:

> **RQ$_2$.** *How does the presence of a security checklist affect vulnerability detection during a security-focused code review?*

We organize **RQ$_2$.** in two sub-questions. First, we ask:

RQ$_{2.1}$. To what extent does the presence of a security checklist affect vulnerability detection during a security-focused code review?

The presence of items irrelevant to the change might deteriorate developers' code review performance. Brykczynski [18] suggests that users are less likely to read through a multitude of checklist items. To mitigate this problem, one could imagine that future

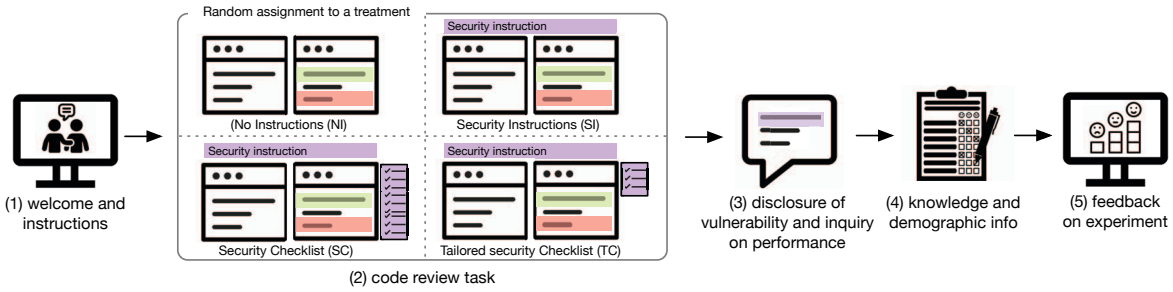

Figure 1: Steps of our online experiment.

automation techniques will be able to generate a checklist *tailored* to the code change to review. We manually create such an ideal checklist to measure its effect and ask:

RQ$_{2.2}$. To what extent does the presence of a *tailored* security checklist affect vulnerability detection during a security-focused code review?

Our formal hypotheses for **RQ**$_{2.1}$ and **RQ**$_{2.2}$ are as follows:

H0$_{2.1}$: The presence of a security checklist does not affect vulnerability detection during a security-focused code review compared to providing security instructions.

H0$_{2.2}$: The presence of a *tailored* security checklist does not affect vulnerability detection during a security-focused code review compared to providing security instructions or a more generic checklist.

3.2 Experimental Design

Figure 1 presents the flow of our online experiment, which consists of five steps. Each step corresponds to one or two different web pages, and our experimental design does not allow the participants to return to previous pages or redo the experiment.

(1) Welcome page: We provide the participants with general information about our study. We indicate that our goal, in general, is to improve code review practices and that they are going to do a code review. We also present our data handling policy to the participants and ask for their consent to use their data.

(2) Code review task: We first ask the participants to take the review task as seriously as possible and to assume that the code they are going to see compiles and all tests pass. Depending on the treatment, participants may receive further instructions:

– **No security Instructions** (**NI**): Participants are provided no additional instruction.

– **Security Instructions** (**SI**): Participants are informed that we are interested in security issues in th)is review. We also provide them with a definition of software vulnerability.

– **Security Checklist** (**SC**): Participants receive the same instructions as in **SI**. In addition, they are informed that they will have acces to a vulnerability checklist to assist them in the review and explain that each checklist item can be answered as yes/no/irrelevant with a specific checkbox. We ask them to work through every item in the checklist.

– **Tailored security Checklist** (**TC**): Participants receive the same instructions as in **SC** but with a strictly tailored security checklist.

Finally, we provide all participants detailed instructions on how to add/edit/delete review comments.

When ready, the participants can press a button confirming that they have read all instructions and want to start the review. Figure 2 shows a snapshot of the code review task that the participants assigned to **SC** received after pressing the aforementioned button.

When the participants are done, they press a complete button. We warn participants assigned to **SC** and **TC** about any unresolved checklist items, to ensure as high as possible checklist interaction. In addition, we ask participants of all four treatments whether they were interrupted during the code review and for how long.

(3) Disclosure of vulnerabilities and inquiry on performance: After they submit their review, we inform participants that the source code contained two vulnerabilities (**MSI** and **BRA**) and show their location, types, definitions, and effects. Then, for each vulnerability we ask whether they found it. We ask participants to reason on why they could or could not detect it. In addition, we ask the participants assigned to the treatments **SC** and **TC** whether the checklist helped them detect the vulnerabilities. We also ask their general feedback about the checklist, including whether and why they skipped any items. This step helps us better understand the impact of the security checklist on vulnerability detection and collect qualitative data for triangulating our quantitative findings.

(4) Security knowledge and demographics: We ask a series of questions designed to gather information about factors that may affect the participants' detection of the vulnerabilities. Questions are mainly about the participants' security knowledge, checklist experience, and team culture. Most questions are in Likert scale format [84]. All questions are in the enclosed material [16]. Finally, we ask a series of demographics questions about the participants' gender, highest education level, employment status, and years of experience in professional software development, Java programming, code reviewing, web application development, and databases. These questions are mandatory to fill in as collecting such data helps us identify which portion of the developer population our study participants represent [31]. We also ask about the frequency with which they designed, developed, and reviewed code in the last year; this helps us investigate possible confounding factors.

3.3 Experimental Objects

The experimental objects consist of the code change to review, the injected vulnerabilities (**MSI** and **BRA**), and the digital checklists provided to participants assigned to the treatments Security

Figure 2: Example of the code review task with a security checklist using the tool.

Checklist (**SC**) and Tailored security Checklist (**TC**). All the material is available in our replication package [16].

Code Change. The code change is implemented in Java: being Java one of the most popular programming languages [82], this allows us to reach a broader population of developers. The change comprises a feature implemented to manage users' online registration to a web service, consisting of two classes, five methods, and 160 lines. To avoid giving some participants advantages over others due to familiarity with the code, our code change does not belong to any existing codebase. Instead, we implemented a code change that is suitable for injecting both vulnerabilities. The code change is self-contained and suitable for being part of an existing software (*e.g.*, not a toy example to teach beginners Java programming). Finally, the code change is sufficient to consider possible attack scenarios, thus detecting the two vulnerabilities.

We implemented the first version of the code change, after several brainstorming sessions. Later, we interviewed two security engineers with five and two years of professional security experience. They performed the experiment with treatment **SC**. We asked them to read the code change before reading the checklist. Both security engineers pointed out the vulnerabilities. We also asked them to provide feedback on the checklist. Finally, we conducted a pilot study (Section 3.5). Based on the feedback we received from the security engineers and the participants of the pilot study, we iteratively modified the code change to ensure its realism and remove any confounding implementation or design-related issues other than the two vulnerabilities.

Security Vulnerabilities. The code change contains two vulnerabilities. The first one is a *Generation of Error Message Containing Sensitive Information* (**MSI**– CWE 209) [60]. When an `SQLException` is raised, the query containing an unencrypted plain text password and the user's email address is output to a log file. The error log

```
try {
    con = DBConnection.createConnection();
    statement = con.prepareStatement("insert into user values(?,?,?,?,?,?)");

    statement.setString(1, firstName);
    statement.setString(2, lastName);
    statement.setString(3, email);
    statement.setString(4, userName);
    statement.setString(5, generateHash(password));
    statement.setString(6, salt);

    int result = statement.executeUpdate();

    if(result!=0) {
        return "SUCCESS";
    }
} catch (SQLException e) {
    String logMessage = "Unable to retrieve account information from database,"
            + "\n query: " + statement;
```

CWE-209: Generation of Error Message Containing Sensitive Information: The log message may contain sensitive information stored in the prepared statement ('statement') it includes. If an error is raised, this sensitive information is output to a log file which in turn can be used to simplify other attacks such as SQL Injection. Furthermore, a password salt separated from the password hash can be exploited using rainbow tables.

(a) *Generation of Error Message Containing Sensitive Information* (MSI)

```
public class PasswordManagement {

    private final static char[] hexArray = "0123456789ABCDEF".toCharArray();
    private final static String HASH_ALGORITHM = "MD5";
```

CWE-327: Use of a Broken or Risky Cryptographic Algorithm: The MD5 algorithm has been designed for speed and has been subject to several security breaches, as it became computationally possible to generate collisions, meaning different messages will lead to the same hash. Due to its fast design, MD5 is also susceptible to brute-force attacks.

(b) *Use of a Broken or Risky Cryptographic Algorithm* (BRA)

Figure 3: Object vulnerabilities used in our experiment.

will contain the user's sensitive information. The second vulnerability is the *Use of a Broken or Risky Cryptographic Algorithm* (**BRA**– CWE 327) [62]. The *MD5* cryptographic hash function has been shown to be vulnerable to collision attacks. Different messages may have the same MD5 hash, making forgery attacks possible [76]. Figure 3 shows the **MSI** and the **BRA** we used in our study.

We based our choice of vulnerabilities on the following criteria: prevalence, recognition, and discoverability using the selected checklist. Therefore, we selected vulnerabilities from the OWASP's "Top 10 Web Application Security Risks" list [64] and we selected

vulnerabilities that one could detect through the popular OWASP checklist [63] we employed (see more details below). After the selection, we asked the two security experts to perform the review (we did not disclose the vulnerabilities). They were able to identify both vulnerabilities and pointed out which items in the checklist can help developers to detect these vulnerabilities. They further confirmed that these vulnerabilities are known and prevalent.

Security Checklists. The official checklist available in the OWASP Code Review Guideline has 78 items covering the most critical security controls and vulnerability areas (including **MSI** and **BRA**). We prepared the checklists for the **SC** and **TC** treatments using the OWASP checklist and improved it by applying the guidance offered by the literature [18, 19, 36, 92]. The literature suggests that a checklist should be concise and no longer than one page for best practice and usability [18, 36, 92]; moreover, the categories, which represent particular features of the application, should guide the reviewers "where to look" in the code [19]. Therefore, we designed the checklist available to participants of **SC** by reducing the OWASP checklist to 22 security items structured in 3 categories and 7 subcategories. The **SC** checklist includes items relevant to the application context and the programming language, including two specific items related to the vulnerabilities in the code change. In addition, this checklist contains items irrelevant to the code change to represents a realistic situation in which not all the items are a perfect fit. Moreover, to facilitate participants' interaction with the checklist, we phrased each checklist item in the form of a question and provided a drop-down for the corresponding answer. Finally, the resulting checklist was used during our interview with two security engineers (Section 3.3 – **Code Change**). They provided feedback on each item of the checklist and pointed out the items that disclosed the vulnerabilities in the code change. To prepare the checklist for the treatment Tailored security Checklist (**TC**), we reduced the checklist to a shorter version containing ideal-case scenario items, *i.e.,* only items relevant to the code change. This shorter checklist that we made available to **TC** participants contains seven security items structured in three categories.

3.4 Variables, Measurements, and Analyses

Table 1 presents all the variables we consider in our experiment. The independent variable (*VulnFound*) measures whether the participants found the vulnerability. The main independent variable is *Treatment* (**NI**, **SI**, **SC** or **TC**). We consider the other variables as control variables, which also include the time spent on the review, the participant's role, years of experience in java, code review, and using checklists during code reviews. Details about interruptions (*Interruptions*) are collected from the participants, and the duration (*DurationExp*) of each review is computed from the experiment's log. To answer $\mathbf{RQ}_1$, we build two multiple logistic regression models, one for each vulnerability (**MSI** and **BRA**) as dependent variable. The models are similar to the one used by Mcintosh et al. [47], Spadini et al. [77], and Braz et al. [17].

To ensure that the selected logistic regression models are appropriate for the data we collect, we (i) reduced the number of variables by removing those with Spearman's correlation higher than 0.5 using the VARCLUS procedure; (ii) ran a multilevel regression model to check whether there is a significant variance among reviewers, but we found little to none, thus indicating that a single level regression model is appropriate; and (iii) we added the independent variables into the model step-by-step and found that the coefficients remained stable.

Table 1: Variables used in the logistic regression models.

Variable	Description
Dependent Variables	
VulnFound	The participant found the vulnerability in the code review
Independent Variables	
Treatment	The treatment to which the participant was assigned
Control Variables (Review)	
Interruptions	For how long the participant was interrupted during the review
DurationReview	Duration of the code review
Control Variables (Security Knowledge)	
Familiarity	Familiarity to vulnerabilities
Courses	The participant has participated in security courses and/or training
Update	The participant keeps themselves up to date with security information
Control Variables (Security Practice)	
Incidents	The participant has experience with security incidents
Responsibility	The participant looks for vulnerabilities as a part of their job responsibility
{Designing/Coding/ Reviewing }	The participant actively considers vulnerabilities when {designing software|coding|reviewing code}
Control Variables (Demographics)	
LevelOfEducation	Highest achieved level of education
EmploymentStatus	Employment status
Role	Role of the participant
OSSDev	The experience in OSS development
{ProfDevExp | JavaExp |ReviewExp | WebDevExp | DBDevExp | ChecklistExp }	Years of experience {as professional developer | in java | in code review | in web programming | in database applications | using checklists during reviews}
{DesignFreq | DevFreq | CRFreq }	How often they {design software | program | review code}

Analysis of code review outcome. To give a value to our dependent variables (*i.e., VulnFound*, whether the participants found the vulnerabilities), we do the following: (i) the first and last authors together inspect a subset of the remarks and checklist items interactions made by the participants during the review task and classify each vulnerability as detected or not; then, (ii) the first author classifies the remaining remarks and checklist items interactions; the first and last authors discuss the classification, especially unclear cases. The final decision is taken by cross-checking our classification with the answers participants gave when indicating whether they found the vulnerabilities (Step 3 in Figure 1).

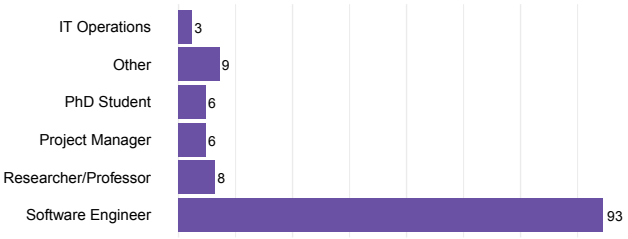

Figure 4: Job distribution among employed participants.

Analysis of open answers on performance. We use open card-sorting [78] to analyze the the open answers participants' gave on their review performance (step 3, Figure 1). It allowed us to identify factors that might affect the detection of vulnerabilities during a review. From the open-text answers, the first and second author separately created self-contained units, then sorted them into themes. To ensure the themes' integrity, the authors iteratively sorted the units several times. Then, both authors compared and discussed their results to reach the final themes. The discussion helped to evaluate controversial answers, reduce bias caused by wrong interpretations of participants' comments, and strengthen the confidence in the outcome. We also use the card sorting output to triangulate our results and form new hypotheses, which we challenged with experimental data (*e.g.,* end of Section 4.2).

3.5 Pilot Runs

We conducted 13 pilot runs to verify (1) the absence of technical errors in the experiment platform, (2) the ratio with which participants were able to find the injected vulnerabilities, (3) the understandability of the instructions and UI, (4) the absence of design/implementation issues beyond the injected vulnerabilities, and (5) the usability of the security checklist. We also gathered qualitative feedback from the participants.

We conducted each run with a different participant. We recruited the participants through our professional network to ensure that they would take the task seriously and provide feedback on their experience. The participants' data and qualitative feedback during the pilot runs were discussed iteratively among the authors every few pilot runs. We continued with our pilot iterations until the required changes were minimal. No data gathered from the pilot is considered in the final experiment.

3.6 Recruiting Participants

To recruit participants, the study authors spread the experiment through direct contacts from their professional network as well as their social media accounts, such as Twitter. In addition, the experiment was spread through practitioners blogs, web forums, and open source mailing lists. The actual aim of the experiment was not revealed. We introduced a donation-based incentive of 5 USD to a charity per complete and valid participant.

4 RESULTS

In this section, we report the results of our investigation.

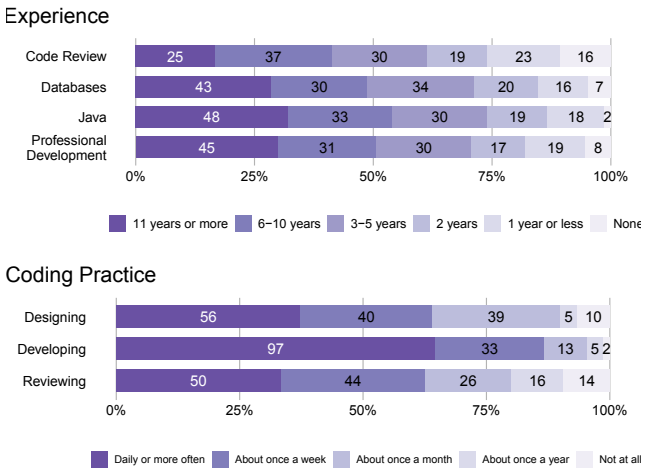

Figure 5: Participants' demographics (absolute numbers).

4.1 Validation of participants data

A total of 522 people accessed our online experiment through the provided link. Of these, 357 did not finish all experiment steps; thus, we removed their entries from the results dataset. We considered the 165 people who completed all steps as potentially valid. Then, we manually analyzed cases of participants whose code review duration was 1.5 times the interquartile range above the upper quartile or below the lower quartile. We removed participants who did not leave any remarks and did not interact with the qualitative questions of the performance inquiry. We also verified the checklist interactions (e.g., if they skipped all items of the checklist) of participants assigned to **TC** and **SC**. We removed 15 participants during this step. After applying these exclusion criteria, data from 150 participants could be used for the analyses.

The valid participants were assigned to the treatments as follows: 33 received **NI**, 41 received **SI**, 41 received **TC**, and 35 received **SC**. Figure 4 shows the current positions of participants with part-/full-time employment, and Figure 5 presents the participants' experience and practice. Most participants currently have an engineering role (62%), have more than two years of professional development experience (82%), and design, program, and review code daily (37%, 65%, and 65%, respectively). 121 participants self-described as male, 7 as female, and 22 preferred not to disclose.

4.2 RQ_1. Security instructions in review

Table 2 presents the results of the review in terms of vulnerability finding: 92 participants found at least one of the vulnerabilities; 37% (55) of the participants found the **MSI**, while 53% (80) found the **BRA**. Among the control participants (**NI**), 3% found the **MSI** and 21% found the **BRA**; among the 117 participants in the treatments approximately 46% found the **MSI** and 62% the **BRA**. When expressed in odds ratio, these results show that participants assigned to any of **SI**, **SC**, and **TC** are eight times more likely to find a vulnerability than those assigned to **NI** ($p < 0.001$).

In our logistic regression models, we set **NI** as the reference level to better see the significance of the treatments (**SI**, **SC** and **TC**) in our *Instructions* variable. We used the same starting variables in both

Table 2: Vulnerability detection by type and treatment.

Treatment	*Vulnerability*	**Found**	**Not Found**
NI– No security Instructions	MSI	1 (3%)	32 (97%)
	BRA	7 (21%)	26 (79%)
SI– Security Instructions	MSI	19 (46%)	22 (54%)
	BRA	25 (61%)	16 (39%)
SC– Security Checklist	MSI	16 (46%)	19 (54%)
	BRA	20 (57%)	15 (43%)
TC– Tailored security Checklist	MSI	19 (46%)	22 (54%)
	BRA	28 (61%)	13 (39%)
Total	MSI	55 (37%)	95 (63%)
	BRA	80 (53%)	70 (47%)

Table 3: Logistic regression models for RQ_1 (N=150).

	Dep. Var. = **MSI**			Dep. Var. = **BRA**		
	Estim.	**S.E.**	**Sig.**	**Estim.**	**S.E.**	**Sig.**
Intercept	-6.080	2.089	**	-5.893	1.741	***
SC	3.809	1.165	**	1.686	0.617	**
SI	4.047	1.152	***	1.627	0.617	**
TC	3.984	1.160	***	1.933	0.631	**
DurationCR	0.040	0.013	**	0.017	0.011	
Interruptions	-0.403	0.177	*	-0.143	0.166	
Familiarity	1.575	0.827	.	0.477	0.692	
Incidents	-0.703	0.602		0.332	0.504	
Practice	0.566	0.498		0.672	0.462	
Update	-0.464	0.302		0.058	0.263	
Responsibility	0.226	0.234		0.068	0.201	
Designing	-0.276	0.321		-	-	-
Coding	0.815	0.326	*	0.192	0.223	
Reviewing	-0.159	0.240		0.176	0.207	
ProfDevExp	-0.017	0.186		0.176	0.207	
JavaExp	0.110	0.194		0.173	0.169	
OftenDesign	-0.195	0.248		-	-	-
DevFreq	-0.132	0.342		0.180	0.254	
CRFreq	0.173	0.222		-0.106	0.198	

Sig. codes: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$

models (see Section 3.4), but the final ones differ due to removals during the multicollinearity analysis. Table 3 shows the results of the logistic regression models considering as dependent variables whether the participants found **MSI** and **BRA**, respectively. The models confirm the result shown in Table 2: Instructing developers to focus on security and providing checklists is significant, thus, *we can reject* $\boldsymbol{H0}_1$.

> ***Finding 1****. Developers who are instructed to focus on security issues during code review are eight times more likely to detect a vulnerability than participants who are not.*

Qualitative Analysis – NI participants. By analyzing the answers **NI** participants gave on why they did (not) identify the vulnerability, we find recurring themes. Considering the case of **MSI**, only one participant found it and explained: "I work in a bank and in my company they send warning emails about this kind of sensitive information being logged (like card numbers being logged etc). So it caught my attention that what is being logged." The top-three reasons participants gave for not detecting **MSI** are they (i) focused on aspects unrelated to security (nine mentions), overlooked the vulnerable code (six), or lacked the necessary knowledge (four). For instance, a participant reported: "To be honest, I was optimizing for code structure/readability/architecture instead of security. Even so, I would probably have missed that one anyways."

Concerning **BRA**, No security Instructions (**NI**) participants reported finding it due to prior knowledge (four mentions) or the explicit use of the *MD*5 algorithm (one). For instance, a participant wrote: "It's very obvious that MD5 is being used to hash the password and MD5 being broken has been known for a long time now." Yet, 79% of the **NI** participants did not find this vulnerability; the top-three reported reasons are: (i) lack of knowledge and experience (ten mentions); (ii) focus on aspects unrelated to security (five); and (iii) wrong assumptions about the code (four). Three participants reported that detecting the vulnerability was not part of their responsibility; as one put it: "I am not very familiar with cryptography and I would assume there would be some separate security assessment."

Qualitative Analysis – SI participants. The **SI** participants received instructions to focus on security issues during the code review. Concerning **MSI** they stated they found it because (i) the code involves user or sensitive information that is valuable to hackers and raises security warning (eight mentions); (ii) of previous knowledge (one); and (iii) of first-hand experience (one). For example, a stated: "It contains personal information which is to be protected according to the GDPR. The leak of the password hash could be a [vulnerability] depending on how [the access] to the logs differs from access to the database." The **SI** who did not detect the **MSI** explained that they missed it because they: (i) overlooked the vulnerability (six mentions), (ii) lacked knowledge or experience (four), and focused on factors unrelated to security (one).

Regarding **BRA**, the **SI** participants mostly reported reasons related to previous knowledge as to why they found it: (i) prior knowledge or experience (seven mentions); the vulnerability is well-known (two); *MD*5 can be hacked and quickly broken (two); and, *MD*5 is an old vulnerability. A participant pointed out: "It's rather well known that MD5 isn't secure anymore so I immediately noticed.' Among the **SI** participants who did not find **BRA** most (seven mentions) reported that they did not detect this vulnerability due to a lack of knowledge or experience. In addition, a participant stated to have made a wrong assumption: "I was expecting that this could be configured in another part of the software."

We challenged these qualitative reasons using data collected in step 5 (Figure 1). We used the variables described in Section 3.4 to map the reasons. We used Chi-Square test for the first two variables of *Knowledge* (*Familiarity* and *Courses*) and Mann-Whitney U test for *KnowledgeUpdate* and all variables of *Practice*. Regarding **MSI**, only *Coding* was significantly related to detecting this vulnerability ($p < 0.01$). On the other hand, all variables of *Practice* (*incidents*, *responsibility*, *design*, *coding*, *reviewing*) were significantly related to detecting **BRA** (all with $p < 0.01$). Additionally, *design* from the *practice* variables group was also significantly related ($p = 0.02$).

Table 4: Logistic regression models for RQ_2 (N=117).

	Dep. Var. = **MSI**			Dep. Var. = **BRA**		
	Estim.	**S.E.**	**Sig.**	**Estim.**	**S.E.**	**Sig.**
Intercept	-1.948	1.832		-6.007	2.102	**
SC	0.244	0.557		-0.016	0.571	
TC	0.167	0.546		0.288	0.569	
DurationCR	0.0425	0.014	**	0.023	0.013	.
Interruptions	-0.402	0.181	*	-0.291	0.190	
Familiarity	1.636	0.843	.	0.619	0.777	
Incidents	-0.535	0.629		0.441	0.635	
Practice	0.681	0.519		0.817	0.545	
Update	-0.505	0.309		0.005	0.304	
Responsibility	0.186	0.241		0.078	0.244	
Designing	-0.344	0.330		-0.033	0.324	
Coding	0.829	0.335	*	0.190	0.262	
Reviewing	-0.124	0.244		0.234	0.245	
ProfDevExp	-0.055	0.195		0.082	0.185	
JavaExp	0.122	0.202		0.182	0.206	
OftenDesign	-0.130	0.251		0.022	0.254	
DevFreq	-0.189	0.355		0.630	0.365	.
CRFreq	0.147	0.230		-0.083	0.230	

Sig. codes: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$

***Finding 2**. Most developers perceive that security knowledge and practice influence their ability to detect vulnerabilities.*

4.3 RQ_2. Security checklists in reviews

In $\mathbf{RQ_2}$ we investigate the effect of providing a security checklist in addition to asking developers to focus on security issues. In this analysis, we thus remove the participants of No security Instructions (**NI**) and set treatment Security Instructions (**SI**) as the reference level for our logistic regression models.

Table 4 shows the results. While *Duration* and *Interruptions* are still significant for the detection of **MSI**, neither **TC** nor **SC** treatments are significant. Therefore, *we cannot reject* $\boldsymbol{H0_2}$. The presence of a security checklist does not affect the detection of software vulnerabilities during code review compared to only asking developers to look for security issues.

***Finding 3**. Developers receiving a checklist (even if tailored to the code change) did not find more vulnerabilities than developers only instructed to focus on security issues.*

Qualitative Analysis – TC and SC participants. We asked **TC** and **SC** participants whether the checklist helped them detecting the vulnerabilities.

Checklist was helpful: Concerning **MSI**, the top-three reasons given by participants about why the checklist helped them finding it are: the checklist (i) marked the participants recheck the code (eight mentions), (ii) pointed out specific areas (seven), and (iii) primed the participants to look for vulnerabilities (three). A participant explained: "Did not see the data passed in the logger at first. But after seeing that item in the checklist I went back to check again." Two participants stated to not need the checklist to find the vulnerability but they still reported it as helpful; one stated: "Even though I found this vulnerability before looking at the checklist, it made me specifically look for that again which is probably a good thing."

Regarding **BRA**, the top-three reported reasons for the checklist being helpful are that the checklist: (i) helped participants to look for vulnerabilities (five mentions), the checklist pointed out specific areas (five), and led them search for further information (external sources) regarding the vulnerability (three). A participant explained: "Did not notice which algorithm was used at first. After seeing the checklist I remembered the issue about the md5 algorithm.' 'Another stated: "While review, I search[ed on] Google how strong MD5 is."

Checklist was not helpful: Some **TC** and **SC** participants who identified the vulnerabilities stated that the checklist did not help. Regarding the **MSI**, the top-four reported answers are: (i) the participant noticed the vulnerability before using the checklist (five mentions); (ii) the participant already looks for this vulnerability (one); and (iii) the vulnerability was easy to spot (one). For instance, a participant reported: "I was already on the lookout for that kind of mistake, because in a past job part of my responsibility was making sure that didn't happen." Regarding the **BRA**, the top-four reported answers are: (i) the participant already knew about this vulnerability (15 mentions), (ii) the checklist was not specific enough (four), (iii) the participant did not pay attention to the checklist (two), and this vulnerability requires explicit security knowledge (two).

Participants in **TC** and **SC** who did not find the vulnerabilities also explained why the checklist did not help. Regarding **MSI**, the top-four answers are: (i) lack of attention (three mentions); (ii) they focused on something unrelated to security (three); and (iii) they overlooked the vulnerability (two). For instance, a participant reported: "I wasn't focused on the exception part, hence I missed the vulnerability." Regarding the **BRA**, most participants reported lack of knowledge (six mentions) as the reason the checklist did not help them detect the vulnerability. A participant explained: "I wasn't aware that MD5 was this vulnerable: I estimated during the review that it was a good choice."

***Finding 4**. Overall, 32 out of 76 of the developers who received a security checklist perceived it as helpful. Yet, ten reported to be able to detect the vulnerabilities without it.*

4.4 Robustness Testing

We employ *robustness testing* [53] to further challenge the validity of our findings.

Finding the first vulnerability distracted the participants. Participants who did not receive security instructions (treatment: No security Instructions–**NI**) might have stopped looking for the second vulnerability after they found the first one assuming they found the only problem. To challenge our results against this hypothesis, we simulated what the results would have been if all participants who detected one vulnerability instead detected both (*i.e.,* they did not stop after finding the first). In the simulation, eight **NI** participants found both vulnerabilities. Using the simulated data, we re-run the analyses and checked the new results. Our logistic regression models achieved the same results as for the original

data: the presence of security instructions (as well as the security checklists) is significant to the detection of both vulnerabilities. Additionally, *Coding* also remains significant. Therefore, even if finding the first vulnerability distracted the participants, this did not impact the final results.

Participants have different levels of experience/practices. One factor that may impact the participants' performance in the review task is their experience and practice (*e.g.,* developers with fewer years of experience might find it more challenging to identify the vulnerability). It is important that the treatment groups are balanced in this aspect. In addition to adding experience and practice as control factors in our regression models, we also performed multiple pairwise-comparison between the means of the treatments using the Tukey HSD test and correcting for multiple tests with the Holm approach. We found no statistically significant difference.

The number participants is too low. To calculate the minimum size sample of our experiment (*i.e.,* number of participants with valid responses), we performed a preliminary power analysis using the G*Power [32], using *Two-tail* test with *odds ratio* = 1.5, $\alpha = 0.05$, *Power* = $1 - \beta = 0.95$, and $R^2 = 0.3$. We used a manual distribution. After running the analysis, we found that our experiment needed a minimum of 143 participants. As our experiment reached 150 valid participants (bigger than necessary), we believe that our sample is representative. However, this sample size is valid only for the first logistic regression model that we built to answer the research question $\mathbf{RQ}_1$. To build the second logistic regression model for $\mathbf{RQ}_2$, we exclude the participants assigned to **NI**. Therefore, for this analysis, we reduced our participants' number to 117, which is still a quite large sample compared to many experiments in software engineering [13]. However, we also conducted other statistical tests to verify the effect of single variables on the expected outcome and reported the results as the sample size could have affected the significativity of the multivariate statistics.

The vulnerabilities are too easy/hard to detect. The vulnerabilities injected in the changes might have affected the validity of our results. Reviewers might not find a too hard to detect vulnerability and get discouraged to continue the experiment, while participants might detect a too easy vulnerability regardless of any other influencing factor, even without paying too much attention to the review. We measure that 37% and 53% of the participants found the **MSI** and the **BRA**, respectively, suggesting that these vulnerabilities were neither too trivial nor too difficult to find.

5 THREATS TO VALIDITY

Internal Validity. As our experiment was online, participants conducted it in different environments (*e.g.,* noise level and web searches), which could have influenced the results; yet it is expected that developers in real world settings also work with various tools and environments. Interruptions could have also influenced the results, so we asked participants about interruptions and their duration and included this information in our statistical analyses. To mitigate the possible threat from missing control over subjects, we included some questions to characterize our sample, such as experience and role (Step 4 in Figure 1). We conducted statistical tests to investigate how these factors affect the results. Participants were allowed to take the experiment only once to prevent duplicate participation. We removed participants who did not complete the experiment and manually analyzed outliers in terms of duration.

We acknowledge that developers' background, knowledge, and practice may impact the experiment's results. However, we could only register a statistically significant (positive) effect for the variable 'coding' (i.e., "the participant considers vulnerabilities when coding" – see table 1) when finding the **MSI** vulnerability (Tables 3 and 4). In their open-text responses, developers indicated that they perceive these aspects (especially knowledge) to play a substantial role. In addition, we performed multiple pairwise comparisons between the means of the treatments using the Tukey HSD test and correcting for multiple tests with the Holm approach (section 4.4). We believe the impact of background knowledge and practice should be investigated with further studies.

Construct Validity. The vulnerabilities were based on the examples of their CWE description [60, 62] and the security checklists were based on the OWASP code review checklist [63]. To mitigate the risk that the code changes and security checklists could have unanticipated characteristics that biased the results (*e.g.,* due to inappropriate quality), we validated them with two security experts (Section 3.3) and through 13 pilot runs (Section 3.5).

The online platform showed all code on the same page, and participants had to scroll down to proceed to the next page of the online experiment. In this way, we aimed to ensure that subjects saw the complete code change. To mitigate the fact that the online reviewing platform may differ from what participants are used to, the experiment tool's interface is designed to be identical to the popular review tool Gerrit [35]. Documentation was also added to make the experiment closer to a real world scenario.

We used different measurement techniques to mitigate *mono-method bias* [23]: we obtained qualitative results by employing card sorting on participants' responses to their performance inquiry (Step 3 in Figure 1). We also used this technique on the feedback of **TC** and **SC** participants on the helpfulness of the security checklist. We ran statistical analyses on variables from participants' answers (Step 4 in Figure 1) and triangulated the qualitative and statistical findings. To mitigate *mono-operation bias* [23], we used more than one variable to measure each construct (*e.g.,* security knowledge, practice). Each of variable (see Table 1) corresponds to a question in the survey on vulnerabilities (Step 4 in Figure 1). To mitigate *interaction of different treatments* [23]: participants were randomly assigned to one of the treatments (**NI**, **SI**, **TC** or **SC**). To test $\mathbf{H0}_1$ (*i.e.,* the effect of a security instructions on vulnerabilities detection), we analyzed responses of all participants. To test $\mathbf{H0}_{2.1}$ (*i.e.,* the effect of a security checklist on vulnerability detection) and $\mathbf{H0}_{2.2}$ (i.e., the effect of a tailored security checklist on vulnerability detection), we considered responses of participants assigned to **SI**, **TC** and **SC**.

We used the qualitative data–collected from **SC** and **TC** ($N = 76$)– to understand developers' perceptions of the checklist's effect. We used the qualitative data from **NI** and **SI** participants ($N = 74$) to get insights into their perceived reasons for (not) finding the vulnerabilities. Therefore, qualitative answers in these two groups (**SC**-**TC**, **NI**-**SI**) refer to two different contexts. We used this design to limit the number of questions (especially those with open-text answers) to prevent overloading our participants and to ensure broader participation in the experiment.

External Validity. To have a diverse sample of participants, we invited software developers from several countries, organizations, education levels, and background; yet, our sample is not representative of all developers. Moreover, further studies should be designed and run to establish the generalizability of our findings with different changes and vulnerabilities.

6 DISCUSSION

We investigated the effects of instructing developers to focus on security issues and of two security checklists on vulnerability detection in code review. We now discuss the main implications and results of our study.

Mental attitude matters. Vulnerabilities can lead to strong negative consequences when they go undetected. Almost every day the media publishes news about successful cyberattacks, showing more and more the urgency and need for advances in the secure software engineering field. Previous studies reported mental attitude as one leading cause of why vulnerabilities are introduced in the code [50, 88, 91] and initial evidence suggests that it may be one of the reasons for not being detected during code review [17].

In line with previous findings [17, 50, 88], our results suggest that developers' mental attitude plays an important role in the detection of software vulnerabilities during code reviews. These results strengthen the questions on the effectiveness of the current development process, including coding and reviewing activities. Organizations may consider incorporating Secure Code Review (SCR) into their development process to create a different approach. SCR is an enhancement to the standard code review practice where the structure of the review process places security considerations, such as company security standards, at the forefront of the decision-making [63]. Further studies need to be designed and carried out to determine how to better design SCR and evaluate its effectiveness.

Interestingly, our results show that security instructions are helpful but the detection of vulnerabilities does not increase when a security checklist is added to the code review. Thus, a *less is more* approach can be adopted to improve the code review process: keeping it as simple as possible with security instructions rather than incorporating advanced or complicated solutions, such as security checklists. Studies can be carried out to investigate how to effectively address security instructions during SCR.

Security checklists are no silver bullets. Checklists are a well-established reading support mechanism often used by individual inspectors for the purpose of preparation [27]. Previous studies [54, 68] found that Checklist-Based Reading effectively finds more issues during code reviews than Ad Hoc Reading. Braz et al. [17] suggested security checklists as a way to improve the code review process. Surprisingly, we found that they do not facilitate the detection of software vulnerabilities compared to just instructing the developers to focus on security issues during the code review, even when the checklist is strictly tailored to the code change.

Our findings provide initial evidence that reviewers may make wrong assumptions, such as assuming that the developer already implemented the change considering the application's security. In fact, checklists may not always be helpful. For example, it can remind the developer to use hashing to protect passwords. However, in practice, hashed passwords cannot be used for challenge-response authentication, negotiating a shared cryptographic session key, or for other such things [14]. Thus, the developer must decide between protecting against a stolen password file or being able to generate a shared session key. Although well-intentioned, a checklist cannot answer this question and, without proper knowledge, developers may take the wrong decision. In our study, developers reported lack of knowledge as one of the main reasons why the checklists did not help in detecting the vulnerabilities. A security checklist could be used as a reminder but not as a substitute for analysis [14].

A possible problem of checklists is that they are often too general and do not sufficiently tailored to a particular development environment [43]. However, our results indicate that a security checklist strictly tailored to the code change does not increase the vulnerability detection in code reviews. This finding raises questions on whether the problem is not the security checklists but how developers perceive them. In fact, a few developers reported to have read the checklist *after* the review or to not consider it at all. Further studies could investigate how to effectively integrate security checklist in tools and processes to support review.

7 CONCLUSION

We investigated whether and to what extent instructing developers to focus on security issues and providing security checklists during code reviews can support the detection of software vulnerabilities. Specifically, we designed and conducted an experiment with 150 participants, which included a code review task and a survey. The code change to review contained two vulnerabilities: *Generation of Error Message Containing Sensitive Information* (**MSI**) and *Use of a Broken or Risky Cryptographic Algorithm* (**BRA**). The participants were randomly assigned to one of the following treatments: No security Instructions (**NI**), Security Instructions (**SI**), Security Checklist (**SC**), or Tailored security Checklist (**TC**). The latter two groups received a security checklist; **SI** received instructions to focus on security issues during the review; and **NI** received neither.

In total, 92 participants found at least one of the vulnerabilities. Those who received the instructions to focus on security issues were at least eight times more likely to detect vulnerabilities. However—surprisingly—adding security checklists did not increase further the vulnerabilities detection during code review compared to only receiving security instructions.

Our findings provide evidence that developers' mental attitude plays a role in detecting software vulnerabilities during code reviews. The effect of the security instructions provides evidence that vulnerability detection could be triggered with proper security considerations, such as security standards for code reviews. Moreover, the role and design of security checklists should be investigated further to establish and improve their effectiveness.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their thoughtful and important comments, which helped improving our paper. The authors also express gratitude to the 150 valid participants who took part in the study, and gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Projects No. PP00P2_170529 and PZ00P2_186090.

REFERENCES

[1] 2020. CWE Top 25 Most Dangerous Software Errors. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html.

[2] 2020. GitLab: Mapping the DevSecOps Landscape - 2020 Survey. https://about.gitlab.com/developer-survey.

[3] A. Ackerman, L. Buchwald, and F. Lewski. 1989. Software inspections: an effective verification process. *IEEE Software* 6, 3 (1989), 31–36.

[4] A. Ackerman, P. Fowler, and R. Ebenau. 1984. Software Inspections and the Industrial Production of Software. In *Proceedings of the Symposium on Software Validation: Inspection-Testing-Verification-Alternatives*. 13–40.

[5] O. Akinola and A. Osofisan. 2009. An Empirical Comparative Study of Checklist based and Ad Hoc Code Reading Techniques in a Distributed Groupware Environment. *arXiv preprint arXiv:0909.4260* (2009).

[6] M. Alam. 2010. Software security requirements checklist. *International Journal of Software Engineering* 3, 1 (2010), 53–62.

[7] H. Assal and S. Chiasson. [n.d.]. Security in the software development lifecycle. In *Proceedings of the symposium on usable privacy and security*. 281–296.

[8] N. Ayewah and W. Pugh. 2008. A report on a survey and study of static analysis users. In *Proceedings of the workshop on Defects in large software systems*. 1–5.

[9] N. Ayewah, W. Pugh, D. Hovemeyer, J. Morgenthaler, and J. Penix. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008), 22–29.

[10] A. Bacchelli and C. Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the International Conference on Software Engineering*. 712–721.

[11] D. Balfanz, G. Durfee, D. Smetters, and R. Grinter. 2004. In search of usable security: Five lessons from the field. *IEEE Security & Privacy* 2, 5 (2004), 19–24.

[12] T. Baum, O. Liskin, K. Niklas, and K. Schneider. 2016. Factors influencing code review processes in industry. In *Proceedings of the international symposium on foundations of software engineering*. 85–96.

[13] T. Baum, K. Schneider, and A. Bacchelli. 2019. Associating working memory capacity and code change ordering with code review performance. *Empirical Software Engineering* 24, 4 (2019), 1762–1798.

[14] S. Bellovin. 2008. Security by Checklist. *IEEE Security Privacy* 6, 2 (2008), 88–88.

[15] B. Boehm and V. Basili. 2001. Software Defect Reduction Top 10 List. 34, 1 (2001), 135–137.

[16] L. Braz, G. Çalikli, and A. Bacchelli. 2022. Data and Material. https://doi.org/10.5281/zenodo.6026291.

[17] L. Braz, E. Fregnan, G. Çalikli, and A. Bacchelli. 2021. Why Don't Developers Detect Improper Input Validation?'; DROP TABLE Papers;–. In *Proceedings of the International Conference on Software Engineering*. 499–511.

[18] B. Brykczynski. 1999. A survey of software inspection checklists. *ACM SIGSOFT Software Engineering Notes* 24, 1 (1999), 82.

[19] Y. Chernak. 1996. A statistical approach to the inspection checklist formal synthesis and improvement. *Transactions on Software Engineering* 22, 12 (1996), 866–874.

[20] C. Chong, P. Thongtanunam, and C. Tantithamthavorn. 2021. Assessing the Students' Understanding and their Mistakes in Code Review Checklists-An Experience Report of 1,791 Code Review Checklist Questions from 394 Students. In *Proceedings of the International Conference on Software Engineering: Software Engineering Education and Training*. 20–29.

[21] M. Christakis and C. Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the international conference on automated software engineering*. 332–343.

[22] J. Cohen. 2010. Modern Code Review. In *Making Software*. O'Reilly, Chapter 18, 329–338.

[23] T. Cook and D. Campbell. 1979. *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin Company.

[24] T. Cook, D. Campbell, and W. Shadish. 2002. *Experimental and quasi-experimental designs for generalized causal inference*.

[25] D. Cruzes, M. Felderer, T. Oyetoyan, M. Gander, and I. Pekaric. 2017. How is security testing done in agile teams? A cross-case analysis of four software teams. In *Proceedings of the International Conference on Agile Software Development*. Springer, Cham, 201–216.

[26] A. Dunsmore, M. Roper, and M. Wood. 2003. The development and evaluation of three diverse techniques for object-oriented code inspection. *Transactions on Software Engineering* 29, 8 (2003), 677–686.

[27] A. Dunsmore, M. Roper, and M. Wood. 2003. The Development and Evaluation of Three Diverse Techniques for Object-Oriented Code Inspection. *Transactions on Software Engineering* 29, 8 (2003), 677–686.

[28] A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner. 2013. An empirical study on the effectiveness of security code review. In *Proceedings of the International Symposium on Engineering Secure Software and Systems*. 197–212.

[29] M. Fagan. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15, 3 (1976), 182–211.

[30] M. Fagan. 2002. Design and code inspections to reduce errors in program development. In *Software pioneers*. Springer, 575–607.

[31] D. Falessi, N. Juristo, C. Wohlin, B. Turhan, J. Münch, A. Jedlitschka, and M. Oivo. 2018. Empirical Software Engineering Experts on the Use of Students and Professionals in Experiments. *Empirical Software Engineering* 23, 1 (2018), 452–489.

[32] F. Faul, E. Erfelder, A. G. Lang, and A. Buchner. 2007. GPower3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences. *Behavior Research Methods* 39 (2007), 175 – 191.

[33] The OWASP Foundation. 2017. *OWASP Foundation*. Retrieved July 18, 2021 from https://owasp.org/

[34] C. Garrison and R. Posey. 2006. Computer security checklist for non-security technology professionals. *Journal of International Technology and Information Management* 15, 3 (2006), 7.

[35] Gerrit. 2020. Gerrit Code Review. https://www.gerritcodereview.com/.

[36] T. Gilb and D. Graham. 1993. *Software inspections*. Addison-Wesley Reading, Masachusetts.

[37] D. Gilliam, T. Wolfe, J. Sherif, and M. Bishop. 2003. Software security checklist for the software life cycle. In *Proceedings of the International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*. 243–248.

[38] P. Gonçalves, E. Fregnan, T. Baum, K. Schneider, and A. Bacchelli. 2020. Do Explicit Review Strategies Improve Code Review Performance?. In *Proceedings of the International Conference on Mining Software Repositories*. 606–610.

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

[40] HackerOne. 2022. HackerOne. https://www.hackerone.com/.

[41] F. Hermans. 2021. *The Programmer's Brain: What every programmer needs to know about cognition*. Manning Publications.

[42] S. Kollanus and J. Koskinen. 2009. Survey of software inspection research. *The Open Software Engineering Journal* 3, 1 (2009).

[43] O. Laitenberger and J. DeBaud. 2000. An encompassing life cycle centric survey of software inspection. *Journal of systems and software* 50, 1 (2000), 5–31.

[44] F. Lanubile and G. Visaggio. 2000. Evaluating defect detection techniques for software requirements inspections. *International SoftwareEngineering Research Network, Report* (2000), 1–24.

[45] M. Mäntylä and C. Lassenius. 2008. What types of defects are really discovered in code reviews? *Transactions on Software Engineering* 35, 3 (2008), 430–448.

[46] G. McGraw. 2004. Software security. *IEEE Security Privacy* 2, 2 (2004), 80–83.

[47] S. Mcintosh, Y. Kamei, B. Adams, and A. E. Hassan. 2016. An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Software Engineering* 21, 5 (2016), 2146–2189.

[48] A. Meneely and L. Williams. 2010. Strengthening the Empirical Analysis of the Relationship between Linus' Law and Software Security. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[49] A. Meneely and O. Williams. 2012. Interactive Churn Metrics: Socio-Technical Variants of Code Churn. *Software Engineering Notes* 37, 6 (2012), 1–6.

[50] A. Naiakshina, A. Danilova, E. Gerlitz, E. von Zezschwitz, and M. Smith. 2019. *"If You Want, I Can Store the Encrypted Password": A Password-Storage Field Study with Freelance Developers*. 1–12.

[51] A. Naiakshina, A. Danilova, C. Tiefenau, M. Herzog, S. Dechand, and M. Smith. 2017. Why Do Developers Get Password Storage Wrong? A Qualitative Usability Study. In *Proceedings of the Conference on Computer and Communications Security*. 311–328.

[52] A. Naiakshina, A. Danilova, C. Tiefenau, and M. Smith. 2018. Deception Task Design in Developer Password Studies: Exploring a Student Sample. In *Proceedings of the Conference on Usable Privacy and Security*. 297–313.

[53] E. Neumayer and T. Plümper. 2017. *Robustness tests for quantitative research*. Cambridge University Press.

[54] R. Oladele and H. Adedayo. 2014. On empirical comparison of checklist-based reading and adhoc reading for code inspection. *International Journal of Computer Applications* 87, 1 (2014).

[55] O. Pieczul, S. Foley, and M. Zurko. 2017. Developer-Centered Security and the Symmetry of Ignorance. In *Proceedings of the New Security Paradigms Workshop*. Association for Computing Machinery, 46–56.

[56] Strategic Planning. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology* (2002).

[57] A. Poller, L. Kocksch, S. Trpe, F. Epp, and K. Kinder-Kurlanda. 2017. Can security become a routine? A study of organizational change in an agile software development group. In *Proceedings of the conference on computer supported cooperative work and social computing*. 2489–2503.

[58] A. Porter, L. G Votta, and V. Basili. 1995. Comparing detection methods for software requirements inspections: A replicated experiment. *Transactions on Software Engineering* 21, 6 (1995), 563–575.

[59] CWE Project. 2013. *CWE Category - Sensitive Data Exposure*. Retrieved July 19, 2021 from https://cwe.mitre.org/data/definitions/1029.html

[60] CWE Project. 2021. *CWE-209: Generation of Error Message Containing Sensitive Information*. Retrieved June 28, 2021 from https://cwe.mitre.org/data/definitions/209.html

[61] CWE Project. 2021. *CWE-311: Missing Encryption of Sensitive Data.* Retrieved August, 2021 from https://cwe.mitre.org/data/definitions/311.html
[62] CWE Project. 2021. *CWE-327: Use of a Broken or Risky Cryptographic Algorithm.* Retrieved June 29, 2021 from https://cwe.mitre.org/data/definitions/327.html
[63] OWASP Project. 2017. *OWASP Code Review Guide 2.0.* Retrieved June 28, 2021 from https://owasp.org/www-pdf-archive/OWASP_Code_Review_Guide_v2.pdf
[64] OWASP Project. 2017. *OWASP Top Ten.* Retrieved May 27, 2021 from https://owasp.org/www-project-top-ten
[65] OWASP Project. 2017. *Sensitive Data Exposure.* Retrieved June 28, 2021 from https://owasp.org/www-project-top-ten/2017/A3_2017-Sensitive_Data_Exposure
[66] P. Rigby and C. Bird. 2013. Convergent contemporary software peer review practices. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 202–212.
[67] P. Rigby, D. German, L. Cowen, and M. Storey. 2014. Peer review on open-source software projects: Parameters, statistical models, and theory. *Transactions on Software Engineering and Methodology* 23, 4 (2014), 1–33.
[68] G. Rong, J. Li, M. Xie, and T. Zheng. 2012. The effect of checklist in code review for inexperienced students: An empirical study. In *Proceedings of the Conference on Software Engineering Education and Training*. 120–124.
[69] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 181–190.
[70] Y. Shin, A. Meneely, L. Williams, and J. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *Transactions on Software Engineering* 37 (2011), 772–787.
[71] R. Shirey. 2000. Internet security glossary.
[72] X. Shu, D. Yao, and E. Bertino. 2015. Privacy-preserving detection of sensitive data exposure. *Transactions on Information Forensics and Security* 10, 5 (2015), 1092–1103.
[73] Ilja Smarjov. 2020. *OWASP Secure coding practices checklist and training: Assessment of effectiveness in a technology company.* Master's thesis. TALLINN UNIVERSITY OF TECHNOLOGY.
[74] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. Lipford. 2015. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 248–259.
[75] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. Lipford. 2018. How developers diagnose potential security vulnerabilities with a static analysis tool. *Transactions on Software Engineering* 45, 9 (2018), 877–897.
[76] A. Sotirov, M. Stevens, J. Appelbaum, A. Lenstra, D. Molnar, D. Arne Osvik, and B. de Weger. 2008. MD5 considered harmful today, creating a rogue CA certificate. In *Proceedings of the Annual Chaos Communication Congress*.
[77] D. Spadini, G. Çalikli, and A. Bacchelli. 2020. Primers or Reminders? The Effects of Existing Review Comments on Code Review. In *Proceedings of the International Conference on Software Engineering*. 1171–1182.
[78] D. Spencer. 2009. *Card sorting: Designing usable categories.* Rosenfeld Media.
[79] Mohammad Tahaei and Kami Vaniea. 2019. A survey on developer-centred security. In *Proceedings of the European Symposium on Security and Privacy Workshops*. 129–138.
[80] T. Thomas, M. Tabassum, B. Chu, and H. Lipford. 2018. Security during application development: An application security expert perspective. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.
[81] C. Thompson and D. Wagner. 2017. A Large-Scale Study of Modern Code Review and Security in Open Source Projects. In *Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering*. 83–92.
[82] TIOBE. 2020. TIOBE-Index. https://www.tiobe.com/tiobe-index/.
[83] S. Turpe, L. Kocksch, and A. Poller. 2016. Penetration Tests a Turning Point in Security Practices? Organizational Challenges and Implications in a Software Development Team. In *Proceedings of the Symposium on Usable Privacy and Security*.
[84] W. Vagias. 2006. *Likert-Type Scale Response Anchors.* Technical Report.
[85] C. Weir, A. Rashid, and J. Noble. 2016. How to improve the security skills of mobile app developers? Comparing and contrasting expert views. In *Proceedings of the Symposium on Usable Privacy and Security*.
[86] C. Weir, A. Rashid, and J. Noble. 2017. I'd Like to Have an Argument, Please: Using Dialectic for Effective App Security. (2017).
[87] Wikipedia. Last accessed August 2021. Yahoo! data breaches. https://en.wikipedia.org/wiki/Yahoo!_data_breaches#Late_2014_breach.
[88] I. Woon and A. Kankanhalli. 2007. Investigation of IS professionals' intention to practise secure development of applications. *International Journal of Human-Computer Studies* 65, 1 (2007), 29–41.
[89] G. Wurster and P. Van Oorschot. 2008. The developer is the enemy. In *Proceedings of the New Security Paradigms Workshop*. 89–97.
[90] S. Xiao, J. Witschey, and E. Murphy-Hill. 2014. Social influences on secure development tool adoption: why security tools spread. In *Proceedings of the Conference on Computer supported cooperative work and social computing*. 1095–1106.
[91] J. Xie, H. R. Lipford, and B. Chu. 2011. Why do programmers make security errors?. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. 161–164.
[92] Y. Zhu. 2016. *Software Reading Techniques.* Springer.