Why Don't Developers Detect Improper Input Validation?

Braz, Larissa ; Fregnan, Enrico ; Çalikli, Gül ; Bacchelli, Alberto

DOI: https://doi.org/10.1109/ICSE43902.2021.00054

Other titles: DROP TABLE Papers

Why Don't Developers Detect Improper Input Validation? '; DROP TABLE Papers; --

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

Enrico Fregnan
University of Zurich
fregnan@ifi.uzh.ch

Gül Çalikli
University of Zurich
calikli@ifi.uzh.ch

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

***Abstract*—Improper Input Validation (IIV) is a software vulnerability that occurs when a system does not safely handle input data. Even though IIV is easy to detect and fix, it still commonly happens in practice.**

In this paper, we study to what extent developers can detect IIV and investigate underlying reasons. This knowledge is essential to better understand how to support developers in creating secure software systems. We conduct an online experiment with 146 participants, of which 105 report at least three years of professional software development experience. Our results show that the existence of a visible attack scenario facilitates the detection of IIV vulnerabilities and that a significant portion of developers who did not find the vulnerability initially could identify it when warned about its existence. Yet, a total of 60 participants could not detect the vulnerability even after the warning. Other factors, such as the frequency with which the participants perform code reviews, influence the detection of IIV. Data and materials: https://doi.org/10.5281/zenodo.3996696

I. Introduction

A software vulnerability is "a design flaw or an implementation bug that allows an attacker to cause harm to the stakeholders of an application" [1]. To avoid vulnerabilities, much effort has been spent on making web applications more secure [2]–[4], and organizations are shifting security to earlier stages of software development, such as during code review [5]. Nevertheless, dangerous vulnerabilities are still routinely discovered [6].

One of the most prevalent and high-risk vulnerabilities to this day is *Improper Input Validation* (**IIV**) [3], [7]. This vulnerability is the root cause of more than half of the top ten vulnerabilities in the CWE Top 25 list [3] and is present when a software system does not ensure that the received input can be processed safely and correctly.

From a software engineering standpoint, an interesting peculiarity of **IIV** is that it does not require hard-core security expertise to be caught and avoided [8]. So, *why do developers not recognize Improper Input Validation*? Answering this is central to understand how to support developers in building more secure software systems.

One possible answer is that most software developers do not possess even a very basic knowledge of software security: Several surveyed security experts believe that less than half of the developers can spot security holes [5]. Another possible answer is the lack of a proper attitude towards security in information systems professionals [9]. Xie et al. [10] suggest that software developers have general knowledge and awareness of software security and point out the gap between developers' security knowledge and their behaviors. According to the authors' findings, developers do not perform secure development due to factors related to their mindset, such as relying on third parties (*e.g.,* security experts), and other phases of the development life cycle (*e.g.,* design phase), besides external constraints (*e.g.,* deadlines, budget, customer demands, regulations). These studies report the *perception* of developers and security engineers, which may or may not reflect the real situation. Further studies are needed to verify and complement these claims.

In this paper, we present the design and execution of an online experiment we devised to investigate to what extent developers can(not) detect **IIV** and the underlying reasons. We hypothesize that the visibility of a traditional attack scenario (specifically, a *SQL Injection* pattern) affects whether developers can detect an **IIV**. Attack scenarios for *SQL Injection* (**SQLI**) are readily available even in textbooks [11], [12] and popular culture [13], whereas for the detection of other **IIV** vulnerabilities, developers need to discover attack scenarios and assess their possibility to occur themselves. Based on the aforementioned previous studies [9], [10], we also investigate the effect of informing developers about the existence of a vulnerability (i.e., prompting) on its detection. Inspired by previous work [14]–[16] on how prompting facilitates secure password storage, we hypothesize that some developers have the knowledge to find an **IIV** but need to be prompted to focus on it. Finally, we also investigate what the developers report as the reasons why they found or not these vulnerabilities.

We set up our experiment as an online study composed of a code review task, a survey, and a repeated review after prompting. We received valid responses from 146 participants, 82.5% of whom report to be software developers, and 105 have three or more years of professional programming experience. Our main findings include: (1) The visibility of an attack scenario greatly facilitates the detection of **IIV**; (2) prompting has an effect on **IIV** detection, yet many participants cannot detect an **IIV** without a traditional attack scenario; (3) security awareness during code development and frequency of code review play a role in the detection of **IIV**.

Based on our findings, we discuss implications and outline future avenues for research and practice.

II. Related Work

The study of software vulnerabilities and their developer-related factors is a growing area of research in the Software Engineering (SE) research community (*e.g.,* [17]–[20]).

Improper Input Validation. Among vulnerabilities, **IIV**s are prevalent and well-known. Scholte *et al.* [21] investigated the evolution of **IIV** vulnerabilities over a decade with a specific focus on **SQLI** and cross-site scripting. The authors found that the *attack surface* of **SQLI** is much smaller than for cross-site scripting. The attack surface is the set of points on the boundary of a system or an environment where an attacker can try to enter, cause an effect on, or extract data from it [22]. Our study aims to investigate the extent to which developers detect **IIV** vulnerabilities. Therefore, we focus on **SQLI**, which has a relatively small attack surface, hence is easier to detect, and another **IIV** vulnerability, namely *Improper Input Validation for specified Quantity Input* (**IVQI**), which also has a small attack surface but does not have such a visible attack scenario.

Why developers do not spot vulnerabilities. Regarding the underlying causes of why developers introduce and cannot recognize security issues, the study by Woon and Kankanhalli [9] is based on the claim that Information Systems (IS) professionals' intention to practice secure development, which somehow relates to their mindset, is a primary cause for vulnerabilities. The authors investigated factors that are likely to impact IS professionals' intention to secure development through a survey conducted with 184 IS professionals. They found that attitude (determined by the usefulness of security practices to the final software product and IS professionals' career as well as subjective norms) significantly impacts the intention to practice secure development. Xie *et al.* [10] reported gaps between developers' knowledge and the actual practices and behaviors as underlying causes for software vulnerabilities. Xie *et al.* point out the developers' mindset as a main underlying cause for software vulnerabilities. We also focus on this aspect and investigate whether warning developers about the existence of a vulnerability in the code enable its detection. Findings by Xie *et al.* [10] are based on semi-structured interviews that authors conducted with 15 developers, a relatively small sample size. We conducted our study with 146 participants, 57% of whom currently work as a developer, and overall 71% of them work as software practitioners. Our study collects and investigates qualitative and quantitative data while participants are in action during code review through an experiment and as data reflecting participants' perspectives that are collected through surveys.

Code Review for Vulnerability Detection. Previous research [23]–[25] found that more code reviews and reviewers have positive effects on secure software development, whereas the results on the Chromium project by Meneely *et al.* [26] contradict these findings. In a study with more than three thousand Open Source Software (OSS) projects, Thompson and Wagner [27] concluded that code review reduces security bugs. Another study on OSS projects [28] found that code review can identify common types of vulnerabilities. On the other hand, Edmundson *et al.* [29] conducted a study to investigate the effectiveness of a security-focused manual code review of web applications containing SQL injections. The authors' results indicate relatively low effectiveness in vulnerability detection. In line with these results, di Biase *et al.* [30] found that approximately only 1% of Chromium's review comments are about potential security flaws. While these studies [27]–[30] mainly focus on the effectiveness of code review on vulnerability detection, our study investigates whether developers can(not) detect vulnerabilities (specifically **IIV**) during code review and the underlying causes with a focus on developers' knowledge and mindset.

III. Methodology

Overall, our research aims to understand to what extent developers can(not) detect **IIV** vulnerabilities and why. We base our study on an online experiment with several steps (summarized by Figure 1) that we devised to collect different types of evidence as well as self-reported data.

A. Research Questions

We structured our study in two main research questions. With the first research question, we investigate the extent to which developers can detect an **IIV** vulnerability as well as the effect of the visibility of a traditional (*i.e.,* textbook) attack scenario for an **IIV** vulnerability on its detection.

> **RQ$_1$.** *Do developers detect Improper Input Validation (**IIV**) vulnerabilities during code review?*

We organize our research question as follows:

RQ$_{1.1}$. To what extent do developers detect **IIV** vulnerabilities during code review?

RQ$_{1.2}$. What is the effect of the visibility of a traditional attack scenario for an **IIV** vulnerability on its detection during code review?

In particular, we test the following hypothesis for **RQ$_{1.2}$**:

H0$_1$: The visibility of a traditional attack scenario for an **IIV** vulnerability does not affect its detection during code review.

Woon and Kankanhalli [9] and Xie et al. [10] argued that the mindset might be the reason why developers do (not) detect vulnerabilities: Developers may not pay attention to vulnerabilities because it is not their normal role/practice. We test this hypothesis for **IIV** vulnerabilities. To do so, we take inspiration from the studies by Naiakshina et al. [14]–[16], who investigated the effects of prompting on the implementation of secure password storage. After participants complete their first review and answer questions about vulnerabilities, we warn them about the existence of a vulnerability in the code they just reviewed and ask them to reconsider their review if they missed it. This way, we investigate whether warning developers (who missed the **IIV**) about a vulnerability's existence affects their ability to detect the **IIV**. Therefore, our second research question is:

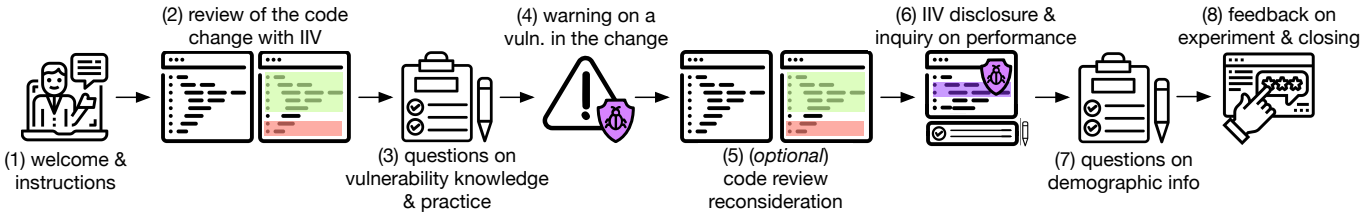

Figure 1. Design and flow of the online experiment.

> **RQ$_2$.** *What is the effect of warning developers who missed the **IIV** about the existence of a vulnerability (i.e., prompting) on the detection of an **IIV**?*

In particular, we test the following hypothesis:

H0$_2$: Prompting does not affect the detection of an **IIV** vulnerability during code review.

B. Design Overview

Our study is implemented as an online experiment that can be reached through a public website. In the following, we detail our study's design and how the experiment flows through each step. Each step corresponds to a different webpage, and returning to previous pages is not allowed.

(1) Welcome page: On the first page of the experiment, we provide participants with information about the study. We introduce ourselves as researchers investigating ways to improve code review. We do not inform the participants about the study's final focus on software vulnerabilities to avoid that they form an interpretation of the purpose of the study and subconsciously change their behavior to fit it (*i.e.,* demand characteristics [31]). We also inform participants about data handling policy and ask for their consent to use their data.

(2) Code review task: In this step, each participant is asked to perform a code review of a code change. The code change includes a method that does not validate the input received from its external users (i.e., **IIV**). There are two slightly different versions (treatments) of this change, and each participant is randomly assigned to either one:

SQLI: The received input is directly used as part of a SQL query making the code vulnerable to *SQL Injection* [32]. The construction of the SQL query is visible in the code, thus showing a traditional attack scenario.

IVQI: The received input includes an integer amount used without validating its boundaries, thus making the code vulnerable to an *Improper Validation of Specified Quantity Input* [33]. Particularly, a negative integer value benefits the users in an unwanted way.

The changed code visible in the review is sufficient to detect both vulnerabilities and how they can be exploited. Both vulnerabilities belong to "Improper Input Validation (**IIV**)" [34].

In addition to the vulnerability (i.e., either **SQLI** or **IVQI**), we introduce an algorithmic bug (a corner case, CC, bug) in the code change. We use this bug as a robustness check to analyze participants' interest in the task. Moreover, immediately after the code review, we ask the participants whether they were interrupted during the review and, if so, for how long.

(3) Knowledge and practice regarding vulnerabilities: In this step, we let the participants know that the study is about security and provide the definition of software vulnerability [35]. We ask questions to gather information about factors that may affect the participants' ability to detect the **IIV** vulnerability (**SQLI** or **IVQI**), such as their security knowledge, practices, and team culture. Most of the questions are closed in a Likert scale format (the exact questions are available in the accompanying material [36]).

(4) Warning about a vulnerability in the change: In this step, we notify the participants that the change they just reviewed has a vulnerability. We do not specify the type of vulnerability to avoid making its discovery too straightforward since we want to focus on the mindset's shift.

(5) Code review reconsideration (optional): Participants are asked to re-inspect the code review unless they think they already found the vulnerability during the first review (hence this step is optional). In the end, we ask participants if they were interrupted during this review and, if so, for how long.

(6) IIV disclosure and inquiry on performance: We show the type and location of the **IIV** vulnerability that affects the change they reviewed and explain how it makes the code vulnerable. Then, we ask the participants whether they found the vulnerability. If not, we ask them to explain why they missed it. If so, we ask in which review they found the vulnerability: If in the first one, we ask them to explain why they could catch it, if in the second one, we ask why they think they could find it only in the second trial. With this step, we aim to check whether the developers agree with our 'miss/found' evaluation of the vulnerability and collect rich qualitative data to triangulate our findings and answer our RQs.

(7) Demographics: Participants are asked to fill in questions to collect demographic information and confounding factors, such as gender, highest obtained education, years of experience (all the questions are available in the accompanying material [36]). This information is mandatory to fill in since collecting such data helps us identify which portion of the developer population is represented by our study participants [37].

(8) Feedback and closing: In the end, we ask the participants for feedback on the overall study. We also ask participants if they would like to share their data anonymously in a public research dataset and receive the study results.

C. Design Implementation

To implement our design, we extend the publicly available browser-based tool CRExperiment [38]. CRExperiment is designed to conduct online experiments that require participants to review code changes and answer survey-like questions; it has been used and validated in previous studies [39], [40]. CRExperiment uses Mergely [41], which is also used by the popular review tool Gerrit [42], to show code changes in two-pane diffs. The Graphical User Interface (GUI) has the same color scheme as Gerrit to facilitate the simulation of a real-world code review scenario during the experiment. In addition to the answers we collect through explicit questions and tasks, CRExperiment also logs user interactions (*e.g.,* mouse clicks and pressed keys), which we use to ensure that participants actively perform the tasks. Finally, CRExperiment logs the time participants spend at each stage of the study. We store all the collected data on a server anonymously. Finally, to reduce the risk of data loss and corruption, we store the data in its raw form (i.e., recorded as logs) for offline analysis.

D. Experimental Objects

The experiment objects are a code change to review and an **IIV** vulnerability (either **SQLI** or **IVQI**) we injected in the code change. We also inject a control bug to use as a robustness check. All the material is available in our replication package [36].

Code Change. Our requirements for designing the object code change are: (1) written in Java, one of the most popular languages [43]; (2) not belonging to any existing code base to avoid giving some developers an advantage over the others due to familiarity with code; (3) suitable for the injection of both vulnerabilities (i.e., **SQLI** and **IVQI**); (4) suitable to be part of an actual software (i.e., not a toy example available on websites aiming to teach beginners Java programming [44]); (5) self-contained. After several brainstorming sessions among the authors, the first version of the patch was implemented. Based on the feedback we received from the pilot studies (Section III-E), we iteratively modified the patch. The final version was discussed and evaluated with two senior software developers with more than ten years of professional software development experience with large software companies. This step led to the last modifications that ensured the change did not have any implementation or design-related issues other than the vulnerabilities and the bug. The change implements a feature to manage employees' vacations, modifying two classes and six methods. The change with **SQLI** has 137 lines of code, while the change with the **IVQI** has 145.

Security Vulnerabilities. We introduced **SQLI** in the code change for the treatment group in the code review experiment, whereas we inject **IVQI** in the code change for the control group. We select **SQLI** to test our hypothesis because it has a stereotyped attack scenario and is presented with a clear pattern in textbooks [11], [12], and even in popular culture [13]. We selected **IVQI** as the vulnerability after several brainstorming sessions among the authors and a final validation with the two aforementioned senior software developers. Even though both **SQLI** and **IVQI** share improper input validation as their root cause and can be neutralized with a solution based on the same principle, the latter does not present a stereotyped attack scenario that can trigger the reviewer's attention. Figure 2 shows the **SQLI** and the **IVQI** used in our online experiment.

```
/**
 * Get the level for an employee, given their employee ID
 *
 * @param employeeID
 * @return the current level of the specified employee
 * @throws SQLException in case of persistence-related issues
 *         (e.g., employee not found)
 */
protected int getEmployeeLevel(String employeeID) throws SQLException {
    String query =
           "SELECT * FROM tblemployees WHERE employeeID='" + employeeID + "'";
```

CWE-89: SQL Injection: Here there is a risk of SQL injection when, for example, an employeeID " or '1'='1'" is used. There are 2 conditions in the query. (1) employeeID = ": It will be evaluated to false as there is no empty employees in the table. (2) '1'='1': It will be evaluated to true as this is static string comparison. Now combining all 2 conditions i.e. false or true => Final result will be true.

```
    ResultSet rs = ConnectionPool.getInstance().executeQuery(query);

    if (!rs.next()) throw new SQLException("Failed to fetch employee");

    int employeeLevel = rs.getInt("employeeLevel");
    rs.close();
    return employeeLevel;
}
```

(a) *SQL Injection* (**SQLI**)

```
/**
 * The the cost of the vacation is based on the employee`s level.
 *
 * @param  numOfDaysToBuy  number of vacation days consumed during
 *         the year
 * @param  employeeLevel current seniority level of the employee
 * @return the holidays to be paid
 */
protected double calculateCostOfVacationDays(int numOfDaysToBuy,
          int employeeLevel) {
    return numOfDaysToBuy *
```

CWE-20: Improper Input Validation: The code does not prevent a negative number from being specified for numOfDaysToBuy. The calculateDaysBetweenDates method will return a negative number if an attacker were to provide a negative number of days to buy, then the employee would be credit instead of debited.

```
}
```

(b) *Improper Validation of Specified Quantity Input* (**IVQI**)

Figure 2. Code snippets with the vulnerabilities in our online experiment.

Control Bug. One of the main reasons developers perform code review is to detect functional defects [45]. We introduce a bug in the object patch as a robustness check to analyze participants' interest in the task. In other words, if we measure that most participants who do not find the bug also miss the vulnerability, the actual cause could be that those participants do not put enough effort into doing the task and, thus, considering their data in the analyses would likely lead to biased results. In such a case, we would only consider participants who found the bug. We inject a Corner Case (CC) bug, which is typically checked by developers [40], as also confirmed by the Google code review guidelines [46] that explicitly encourage developers to check for this type of bug.

E. Pilot Runs

Once the first version of the online experiment was ready, we conducted pilot runs to (1) verify the absence of technical errors in the online platform, (2) check the ratio with which the participants were able to find the injected vulnerabilities (regardless of the treatment group), (3) verify the understandability of the instructions, survey questions, as well as the user

interface, (4) improve the code review tool features to ensure that participants' code review experience is as close as possible to an actual one, (5) verify that the code change does not have any design or implementation issues except for the injected vulnerabilities (either **SQLI** or **IVQI**) and the CC bug, and (6) gather further qualitative feedback from the participants.

We conducted pilot runs for a total of nine participants. 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. The participants for pilot runs were recruited through the authors' professional network to ensure they would take the task seriously and provide detailed feedback about their experience. In the final study, we used no data gathered from any of the participants who took part in the pilot runs.

F. Variables, Measurement Details and Analyses

Analysis of the First Code Review Outcome ($\mathbf{RQ}_1$). To answer $\mathbf{RQ}_{1.2}$, we build a multiple logistic regression model, similar to the one used by McIntosh et al. [47]. The binary dependent variable of our model $VulnFound$ indicates whether the participant detects the vulnerability during the first review or not (Table I). To compute the value of $VulnFound$, we do the following: (1) the second author inspects all the remarks participants made during the code review experiment and classifies each remark as detection of the vulnerability or not, then (2) the first author goes through most of the data together with the second author to discuss the decisions, especially the cases that the second author marks as unclear. These authors take the final decision cross-checking their opinion with the answers participants gave to the corresponding question in Step 6 (Figure 1).

To ensure that the selected logistic regression model is appropriate for the data we collect, we (1) reduced the number of variables by removing those with Spearman's correlation higher than 0.5 using the VARCLUS procedure, (2) further tested for multicollinearity computing the Variance Inflation Factors (VIF) and removing all values above 7, thus ending with little or no multicollinearity among the independent variables, and (3) built the models adding the independent variables step-by-step and found that the coefficients remained relatively stable, thus further indicating little interference among the variables.

The independent variable $VulnType$ (**SQLI** or **IVQI**) is included in the model to investigate how the visibility of a traditional attack scenario for an **IIV** vulnerability affects its detection. To answer $\mathbf{RQ}_{1.2}$, we also need to consider the effect of possible confounding factors related to *security knowledge*, *practice*, and *team culture* on the outcome of the code review experiment (i.e., $VulnFound$). For this reason, we also include in our model a number of control variables (Table I). Values for all these variables (except for the ones that regard the review, such as $BugFound$ and $Interruptions$) are collected through the survey questions in Steps 3 and 7 (Figure 1). Details about interruptions ($InterruptionsFirst$ and $InterruptionsNext$) are collected from the participants at the end of the reviews, and the duration of each review is computed from the experiment's log.

Table I
VARIABLES USED IN THE LOGISTIC REGRESSION MODELS.

Metric	Description
Dependent Variables	
VulnFound ($\mathbf{RQ}_1$)	The participant found the vulnerability in the first code review
VulnFound2 ($\mathbf{RQ}_2$)	The participant found the vulnerability during revisit to the code review
Independent Variables	
VulnType	Type of the **IIV** vulnerability in the code change (**SQLI** or **IVQI**)
Control Variables (Review)	
BugFound	The participant found the functional bug
Interruptions	For how long the participant was interrupted during the review
DurationReview	Duration of the code review
Control Variables (Security Knowledge)	
Familiarity	Familiarity to vulnerab.
Courses	The participant has participated in security courses and/or training
KnowledgeUpdate	The participant keeps himself/herself up to date with security information
Control Variables (Security Practice)	
Incidents	The participant has experience with security incidents.
Responsibility	The participant looks for vulnerab. as a part of his/her job responsibility
StaticAnalysis	How often the participant found static analysis tools helpful in finding vulnerabilities
DynamicAnalysis	How often the participant found dynamic analysis tools helpful in finding vulnerabilities
ManualAnalysis	How often the participant found manual analysis helpful in finding vulnerabilities
{Design/Coding/ Reviewing }	The participant actively considers vulnerabilities when {designing softwarelcodinglreviewing code}
Control Variables (Team Culture)	
{ToolUsage | ThirdPartyLib | CRusage | EnoughTime }	The extend to which developers in the team {use tool to detect vulnerabilities | check for vulnerabilities in third party libraries | use code review to detect vulnerabilities | have time to consider security aspects}
Control Variables (demographics)	
Gender	Gender of the participant
LevelOfEducation	Highest achieved level of education
EmploymentStatus	Employment status
Role	Role of the participant
OSSDev	The experience in OSS development
ReviewPractice	How often they perform code review
{ProfDevExp | JavaExp |ReviewExp | WebDevExp | DBDevExp }	Years of experience {as professional developer | in java | in code review | in web programming | in database applications}
{DesignFreq | DevFreq | CRFreq }	How often they {design software | program | review code}

Analysis of the Review Reconsideration ($\mathbf{RQ}_2$). We build a second multiple logistic regression model to answer $\mathbf{RQ}_{2.2}$. The independent and control variables of the second model are the same as those of the first model we build to answer $\mathbf{RQ}_{1.2}$, whereas the dependent variable is $VulnFound2$ (see Table I). The second model is built using data of participants who did not find the vulnerability during the code review task in Step 2 (Figure 1). We used the same approach as for the regression in RQ_1 to ensure that the selected model was appropriate.

Analysis of open answers on performance. To analyze the answers that participants gave to the open questions when reflecting on the reason for their performance (Step 6, Figure 1), we used open card sorting [48]. This allowed us to extract emerging themes reported as affecting the detection of an **IIV** vulnerability. From the open-text answers, the second author created self-contained units, then sorted them into themes. To ensure the themes' integrity, the author iteratively sorted the units several times. After review by the first author and discussions, we reached the final themes. The discussion helped us evaluate controversial answers, reduce potential bias caused by a wrong interpretation of a participant's comment, and strengthen the confidence in the card sorting process's outcomes. The card sorting supported us in triangulating our results and form new hypotheses that we challenged with experimental data (*e.g.,* end of Section IV-B). The card sorting output is available in our replication package.

G. Recruiting Participants

The online study was spread out through practitioners' web forums, IRC communication channels, direct authors' contacts from their professional networks, as well as their social media accounts (e.g., Twitter, Facebook). We did not reveal the actual aim of the experiment. We also introduced a donation-based incentive of 5 USD to a charity per participant with a complete and valid experiment.

IV. Results

In this section, we describe how we validated the set of participants and report the study results by research question.

A. Valid Participants

A total of 472 people accessed the welcome page of our study's web tool through the provided link. Only 194 people went beyond that page and were considered for the experiment.

From these, we excluded instances in which all study steps were not completed, or the first code review (Step 2) was skipped or skimmed (we checked that at least one remark was entered). We manually analyzed the cases of participants who spent less than one-third of the interquartile range in their code review or more than three. Among these, we detected participants who declared to have not done the task seriously and who said they were interrupted significantly during the code review, so their results could not be completely trusted (from this, we removed 10 participants). After applying the aforementioned exclusion criteria, we had a total of 146 valid participants.

In total, 80 valid participants received the code change with the **SQLI**, and 66 received an **IVQI**. We compared the characteristics of the participants assigned to the two groups and found no statistically significant difference.

In the open-text gender question, 109 and 7 participants self-described as males and females, respectively, and 30 participants preferred not to disclose. The majority of the participants are currently software developers (57%) and reported to have multiple years of experience in professional software development: 23% have 3-5 years of experience, 32% have 6-10 years, and 18% have more than 11 years. Most respondents design, program, and review code daily (61%, 96%, and 64%, respectively). Figure 3 shows the current positions of participants with part-/full-time employment and Figure 4 presents the participants' experience and practice.

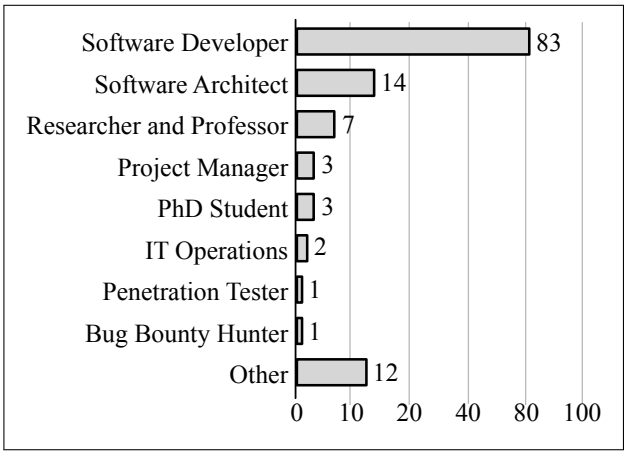

Figure 3. Job of participants with employment.

B. RQ_1. Detecting *IIV* vulnerabilities during code review

To investigate our first research question, we asked participants to review a code change containing a vulnerability (either an **SQLI** or an **IVQI**) and a CC bug.

A total of 76 participants found the CC bug during the first code review, while none of them reported it during the review reconsideration. Among the participants assigned to the code containing the **SQLI**, 39 (49%) found the CC bug. Among those assigned to the code with the **IVQI**, 37% (56%) found the bug. The difference between these groups is not statistically significant, $\chi^2(1, N = 146) = 0.77, p = 0.379$.

Table II
Detection of the vulnerability in the first review (Step 2).

IIV	SQLI	IVQI	Total
Found	52	14	66
Not Found	28	52	80
Odds Ratio: 6.90 (3.27,14.57)			
$p < 0.001$			

Table II presents the results of the code review task (Step 2 in Figure 1) by vulnerability type (**SQLI** vs. **IVQI**). During this step, a total of 66 participants found the vulnerability to which they were assigned. Nevertheless, this number is unbalanced. Out of the 80 participants assigned to **SQLI**, 65% found the vulnerability during the review task (Step 2, Figure 1). On the other hand, 66 participants were assigned to **IVQI**, and 18% found the **IIV** in this step. In this review, 45 participants found neither **SQLI** nor **IVQI**. Expressed in odds ratio, these results show how **SQLI** is seven times more likely to be found by participants than **IVQI** ($p < 0.001$).

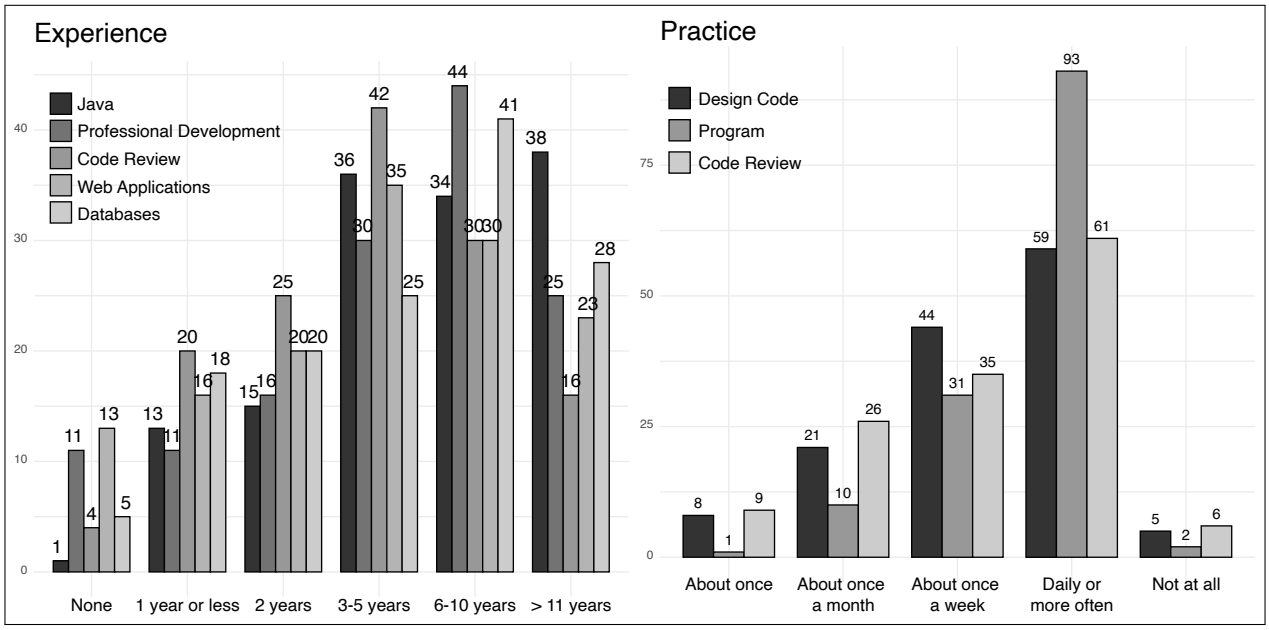

Figure 4. Participants' demographics.

Table III shows the result of the logistic regression model. This statistical model confirms the result shown in Table II: Vulnerability type is significant; thus, *we can reject* $\boldsymbol{H0}_1$.

The other statistically significant variables are (i) $Coding$ (the developer actively considering vulnerabilities when coding) and (ii) $CRFreq$ (how often they review code in the last year). All three significant variables have positive estimate values, which means the higher the values of these variables, the more likely the vulnerability is found.

Table III
REGRESSION FOR THE FIRST CODE REVIEW (STEP 2).

	Estimate	S.E.	Sig.
Intercept	-12.814	3.324	***
VulnType	2.351	0.574	***
BugFound	0.325	0.565	
Interruptions	-0.104	0.244	
Familiarity	0.954	1.247	
Courses	0.242	0.604	
KnowledgeUpdate	0.124	0.404	
Incidents	-0.164	0.703	
Responsibility	-0.388	0.323	
ManualAnalysis	-0.560	0.317	
Coding	1.079	0.422	*
Reviewing	0.625	0.437	
ThirdPartyLib	0.108	0.266	
CRusage	-0.372	0.344	
Role	0.119	0.119	
OSSDev	0.104	0.691	
DBDevExp	0.168	0.271	
DevFreq	0.374	0.465	
CRFreq	0.963	0.418	*
... (†)			

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

(†) $DurationReview$, $StaticAnalysis$, $ToolUsage$, $EnoughTime$, $ProfDevExp$ and $JavaExp$ are not significant and omitted for space reason

By analyzing the answers participants gave on why they identified the **IIV** in the first code review, we find recurring themes. In the case of **SQLI**, the top-three reported reasons are: (i) it is a common and easy to notice vulnerability (18 mentions); (ii) participants have experience with this type of vulnerability (13 mentions); and (iii) participants have knowledge about it (9 mentions). These results reflect our expectations: **SQLI** is a traditional attack scenario and, therefore, easier to be recognized. Participants also reported possessing knowledge and previous experience with **SQLI**: *e.g.,* having exploited the vulnerability. For instance, a participant reported: "I know about SQL-injection, and it is still very high on the OWASP Top 10 list, thus whenever I see an SQL statement, I consider the possibility of injection;" while another stated: "This is literally school example of SQL Injection with hardcoded SQL and concating parameter." However, **SQLI** still frequently happens in practice. For instance, a participant explained: "SQL Injection issues are among the most common and glaring issues in code that I review."

For **IVQI**, developers reported that they follow review practices (10 mentions) to detect this issue: *e.g.,* checking user inputs. A participant reported the following reason: "I always consider function inputs and outputs, especially if user-provided." Moreover, the reasons reported by the participants show that they do not detect **SQLI** and **IVQI** the same way, while their root cause is the same, which supports our aforementioned findings.

We challenged these qualitative reasons using data collected in Steps 3 and 7. We used the variables described in Section III-F to map the reasons. All the $knowledge$ variables ($Familiarity$, $Courses$, and $KnowledgeUpdate$) are correlated with participants finding vulnerabilities of this type. We used *Chi-Square* test for the first two and *Mann-Whitney U* test for the last and obtained $p = 0.03$, $p = 0.03$, and $p = 0.01$, respectively. Regarding the participants assigned to **IVQI**, we

performed *Mann-Whitney U* test on the $practice$ variables and only $Dynamic$ was significantly related ($p = 0.02$).

> ***Finding 1***. *Developers are seven times more likely to detect an **IIV** when a traditional attack scenario is visible (**SQLI**) than when it is not (**IVQI**). Other contributing factors are related to knowledge and practice.*

C. RQ_2. Detecting *IIV* after being warned

To investigate the second research question, we warned the participants about a vulnerability in the code change and invited them to reconsider their review (Step 5 in Figure 1) if they had not found it yet. In this step, no participant found the CC bug. Table IV presents the results for the reconsidered review. In total, 20 additional participants identified the vulnerability to which they were assigned.

We performed a McNemar's test to investigate the effect of warning the participants about a vulnerability in the code change. We considered the code review experiment output for the test. As output, we found that the probability of success is 0.25 with a $p = 2.152e^{-05}$; considering both **SQLI** and **IVQI**, the probabilities of success for **SQLI** and **IVQI** are 0.46 and 0.13 with $p < 0.001$ and $p = 0.02$, respectively. Therefore, *we can reject* $\boldsymbol{H0_2}$. Prompting affects the detection of **IIV**.

In total, 13 participants found the **SQLI**. This means that 46% of the participants who did not find this vulnerability in the first code review found it after the warning. Moreover, 7 participants found the **IVQI**. When expressed in odds, these results show that—when the developers are informed about the existence of a vulnerability in the code (*i.e.,* prompted)—the **SQLI** vulnerability is six times more likely ($p < 0.001$, Table IV) to be found than **IVQI**. This result is in-line with that of our first research question, where we also identified that developers are more likely to detect **SQLI**.

Table IV
ODDS RATIO FOR DETECTING THE VULNERABILITY IN THE REVIEW RECONSIDERATION (STEP 5).

IIV	SQLI	IVQI	Total
Found	13	7	20
Not Found	15	45	60
Odds Ratio: 5.57 (1.88,16.55)			
			$\boldsymbol{p < 0.001}$

In Table V, we show the result of our second logistic regression model. We built the model taking into account only the data of the participants who missed the vulnerability during the first code review. The starting variables used for this statistical model are the same as those used for the first one (see Section III-F), but the final ones differ due to multicollinearity analysis. This model confirms the result shown in Table IV: the vulnerability type significantly affects on its detection during the review reconsideration.

Regarding **SQLI**, participants reported that they found the vulnerability in the review reconsideration for the following reasons: (i) they needed to be reminded to focus on security (13 mentions) or (ii) they lacked the confidence (3 mentions). Some participants reported more than a reason in their answer. The first reason is related as participants needed to pay more attention to the security aspect of the code to identify vulnerabilities; as one participant put it: "At first I was looking mainly for good programming practices and didn't really mind for other issues." The last reason refers to participants who found the vulnerability in the first code review, but were initially not confident if it was indeed an issue; later, the warning clarified their doubt.

Table V
REGRESSION FOR THE RECONSIDERED REVIEW (STEP 5).

	Estimate	S.E.	Sig.
Intercept	-15.261	7.427	*
VulnType	5.584	1.770	**
BugFound	1.164	1.330	
DurationRevisit	0.100	0.062	
Interruptions	-0.409	0.614	
Familiarity	1.934	1.905	
Incidents	-0.186	1.572	
Courses	-0.477	1.476	
KnowledgeUpdate	1.319	0.764	
Responsibility	-0.690	0.624	
Coding	0.633	0.746	
Reviewing	0.735	0.997	
ManualAnalysis	-0.305	0.579	
ToolUsage	-0.144	0.637	
CRusage	0.355	0.636	
ThirdPartyLib	0.274	0.508	
EnoughTime	-0.553	0.540	
OSSDev	0.485	1.569	
ProfDevExp	0.272	0.482	
JavaExp	0.482	0.640	
DevFreq	0.742	0.806	
CRFreq	-0.648	0.606	
... (†)			

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

(†) $StaticAnalysis$ *is not significant and omitted for space reason*

Regarding the participants assigned to **IVQI**, they reported similar reasons: (i) they were looking for non-vulnerability defects (3 mentions) or (ii) they needed to be warned about vulnerabilities (2 mentions). Similar to participants assigned to **SQLI**, these reported focusing on security when they missed the vulnerability in the first code review and needed to be reminded about security issues to start to look for them (*e.g.,* "... in the second review I was actively looking for security flaws. In the first review I was more inclined to look for code correctness"). We see that most participants may know how to identify the **SQLI** correctly and some to detect the **IVQI**, but they tend not to focus on security during a review.

We challenged the reported reasons using data collected in Steps 3 and 7. For the **SQLI**, we related the participants' reasons with variables from $knowledge$ and $practice$ (see Section III-F). We considered the participants that missed the vulnerability in the first code review (*i.e.,* they found it in the review reconsideration or completely missed it). We found no significance for neither the $knowledge$ nor the $practice$ variable categories using *Chi-Square* and *Mann-Whitney U* tests.

***Finding 2**. Prompting has an effect on the detection of **IIV** and most reviewers can detect **SQLI**. Yet, a substantial amount of them cannot detect **IVQI** even after warned of the existence of a vulnerability.*

A total of 60 participants (41%) missed the vulnerability we introduced in the code change. Respectively, 45 and 15 participants did not detect **IVQI** and **SQLI**.

Regarding **SQLI**, developers reported that they missed this vulnerability even after being prompted because they: (i) lacked knowledge or experience (7 mentions), (ii) overlooked the details (2 mentions), or (iii) reviewed different aspects of the code (2 mentions). Although **SQLI** is well-known, some participants still did not know it. For example, a participant reported: "I didn't know that this was a vulnerability." They also reported to be looking for other details in the code (*e.g.,* "I did not think of the database operation in a detailed way, a mistake of mine."). These results highlight the need for better security training, even for basic vulnerabilities, and for improving the development process, so security is also considered, especially during code review.

Regarding **IVQI**, developers reported that they missed this vulnerability because they: (i) lacked attention (10 mentions), (ii) focused on different aspects of the code (9 mentions), (iii) thought that the CC bug was the vulnerability (8 mentions), and (iv) lacked knowledge or experience (6 mentions). The reported reasons support developers' belief in their lack of knowledge or experience to detect these vulnerabilities (13 mentions in total), which is in accordance with what SSEs claim [5]. This reason may be more frequent if we consider "found a different problem" as lack of security knowledge as the developers who reported this reason found the algorithmic bug instead of a security issue.

We challenged these reasons using the data collected in Steps 3 and 7 (in Figure 1). For the **SQLI**, we investigated the $knowledge$ and $practice$ variables categories (seeSection III-F), and found that the following variables are significant: $KnowledgeUpdate$ ($p = 0.01$), $ManualAnalysis$ ($p = 0.49$), $CodeReviewing$ ($p = 0.01$), $Coding$ ($p = 0.01$), and $Design$ ($p = 0.02$). Regarding the **IVQI**, we also investigated the $knowledge$ and $practice$ variables categories. We found that most variables are statistically significant. The not significant ones are: $Familiarity$, $Incidents$, and $Practice$. Our results indicate that both $knowledge$ and $practice$ may be the cause of missing vulnerabilities.

***Finding 3**. Most factors related to low knowledge and practice contribute to missing **IIV** vulnerabilities during code review, even after prompting.*

D. Robustness Testing

To challenge the validity of our findings, we employ robustness testing [49]. For this purpose, we test whether the results we obtained by our baseline model hold when we systematically replace the baseline model specification with the following plausible alternatives.

The functional defect distracted the participants. A significant number of participants reported that they missed the vulnerability because they searched or found other defects. To verify this claim, we checked the correlation between finding the algorithmic bug and finding the vulnerability. We performed a *Chi-square* test considering all participants, only participants assigned to **SQLI**, and only participants assigned to **IVQI**. Our analysis did not achieve a statistically significant value for all three cases. Therefore, we do not have enough evidence to suggest a relationship between finding the algorithmic bug and the vulnerability.

Vulnerabilities were too easy or too hard to find. Choosing the right vulnerability to inject in the code change is fundamental to the validity of our results. If a vulnerability is too easy to find, participants might find the issue regardless of any other influencing factor, even without paying too much attention to the review (on the other hand, if it is too complicated, reviewers might not find any vulnerability and get discouraged to continue). On the one hand, in our study, we aimed to evaluate whether developers can identify a standard textbook vulnerability (**SQLI**). Therefore, we expected this vulnerability to be easy-to-catch. In fact, 81% of the participants found it. On the other hand, **IVQI** is a simple vulnerability to find and fix, but not so recognizable. We measure that 40% of the participants found this kind of vulnerability, thus ruling out the possibility that this vulnerability was either too trivial or too difficult to find.

Number of participants. We performed a preliminary power analysis using the software package G*Power [50] to calculate the minimum sample size (*i.e.,* number of participants with valid responses) for our study. Our prior analysis revealed that we need a minimum sample size of 143 by 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. As our number of participants (146) is bigger than necessary, we believe that they are representative. However, this sample size is valid only for the first logistic regression model that we built to answer the research question $\mathbf{RQ}_{1.2}$. To build the second logistic regression model for the review revisit analysis, we exclude the participants that already found the vulnerability in the code review experiment. Therefore, for this analysis, we reduced our participants' number to 80. Even though this number is quite large in comparison to many experiments in software engineering [51], it could have affected the significativity of the multivariate statistics; for this reason, we also conducted other statistical tests to verify the effect of single variables on the expected outcome and reported the results.

V. Threats to Validity

Construct Validity. The code changes we used in our study are a threat to construct validity. For mitigation purposes, the first and last authors prepared code changes and injected the

vulnerabilities and the corner case bug. The other authors later checked the produced code. To ensure that participants saw the complete code change, the online platform showed all code on the same page on reasonably sized screens; moreover, participants had to scroll down to proceed to the experiment's next page.

A major threat is that online experiments could differ from a real-world scenario. We mitigated this issue by (1) re-creating a code change as close as possible to a real one (*e.g.,* submitting documentation together with the production code), (2) using an interface that is identical to the popular Code Review tool Gerrit [42], (3) injecting vulnerabilities that are based on the examples in the CWE description of both types (**SQLI** and **IVQI**), and (4) getting the code change validated by two professional software developers.

To mitigate *mono-operation bias* [52], we used more than one variable to measure each construct (*e.g.,* security knowledge, practice). Each of these variables (see Table I) correspond to a question in the survey on vulnerabilities (Step 3 in Figure 1). To mitigate *mono-method bias* [52], we used different measurement techniques: We obtained qualitative results by employing card sorting on participants' feedback about why they missed or found the **IIV** vulnerability during the code review task (Step 2, Figure 1) and code review reconsideration (Step 5). We triangulated these qualitative findings with statistical analyses of variables that we obtained through participants' answers to questions in the survey on vulnerabilities. Finally, to mitigate the *interaction of different treatments* [52], we applied each treatment separately as follows: (1) Participants were randomly assigned to one of the treatments: **SQLI** or **IVQI**. We analyzed only the responses participants gave for the code review task (Steps 2) to test the hypothesis $\mathbf{H0}_1$ (i.e., the effect of the visibility of an attack scenario for an **IIV** vulnerability on its detection). (2) To test the hypothesis $\mathbf{H0}_2$ (i.e., the effect of informing participants about the existence of a vulnerability on the detection of **IIV** vulnerability), we analyzed responses that participants who missed **IIV** vulnerability during the code review task gave for the code review reconsideration.

Internal Validity. We reviewed each participation log to identify participants who did not take the experiment seriously. We removed participants who took less than five minutes to complete the experiment or did not complete it. We also introduced a CC bug as a control in the code change for both treatments (**SQLI** and **IVQI**), as explained in Section III-D. We also checked whether the control bug distracted the participants from finding the vulnerability by conducting a *Chi-square* Test of Independence for all participants, only participants assigned to **SQLI**, and only participants assigned to **IVQI**—we did not achieve any significant statistical outcome.

As our experiment was online, we cannot ensure that all participants completed it with the same setup (*e.g.,* monitor resolution) and similar environments (*e.g.,* noise level, interruptions). However, developers in real life also work with different tools in various environments. To mitigate the threats that interruptions might pose to the validity of our study, we asked participants to inform us about durations of interruptions during the code review task and code review reconsideration (Steps 2 and 5 in Figure 1) if there were any. We included these interruptions' durations in our statistical analyses. In addition, several background factors (*e.g.,* age, gender, experience, education) may impact the results. Hence, we collected all such information and investigated how these factors affect the results by conducting statistical tests. Furthermore, we designed our experiment as a within-subject study to reduce random noise due to participants' differences and to obtain significant results with fewer participants [53], [54].

External Validity. We invited developers from several countries, organizations, education levels, and backgrounds. Nevertheless, our sample is certainly not representative of all developers. Thus, further studies are needed to establish the generalizability of our results.

A replication with different vulnerabilities could lead to similar observations as long as they have a similarly popular attack scenario because its effect was clear-cut.

Our observations might not hold when developers review changes to the software projects they work on as a part of their daily practices since higher stakes increase attentiveness [55]. However, some participants mentioned that they never consider vulnerabilities.

Moreover, our results may not be the same if large change-sets or changes that address more than one issue are used in the code review experiment: These are more difficult to review as they increase the reviewer's cognitive load [51]. Therefore, further studies are necessary to assess the generalizability of our results in these scenarios.

VI. Discussion

In this section, we first present themes that emerged as relevant in our study, then provide a high-level overview of the main contributions of our work to research and practice.

A. Emerging Themes

Lack of security knowledge. Software vulnerabilities, such as **IIV**, may have a strong negative impact on software systems, possibly reaching users and even their personal lives. Therefore it seems reasonable to think that developers have the knowledge, training, and practice to make sure vulnerabilities do not reach production systems. However, security experts believe that less than half of developers can actually detect vulnerabilities [5]. Previous studies [9], [10] reported that developers' intention to practice secure coding, general security knowledge, and awareness is the cause of vulnerabilities.

We found that the existence of a visible attack scenario facilitates the detection of **IIV**. Developers struggle to recognize vulnerabilities when such a scenario is not available. Indeed, participants were seven times more likely to find **SQLI** than **IVQI** since there is no popular example of **IVQI** in practice. Furthermore, many participants reported the lack of knowledge and practice as one of the main reasons for not identifying the vulnerability.

In-line with previous findings [5], [10], our results suggest the need to improve developers' security knowledge, but they also call for creating different educational approaches. As attack scenarios seem to be more memorable than generic indications on what should be checked and how, educators may focus more on practical scenarios when teaching security. How to design memorable yet effective and recognizable scenarios is an open research question whose answer can have important practical implications.

Security is not developers' prime concern. Developers reported focusing on other kinds of defects and aspects of the code (*e.g.,* code quality) as one of the main reasons for not identifying the vulnerability. Indeed, our findings highlighted that prompting developers in searching for a security issue had a significant effect on vulnerability detection. Security awareness during code development and the frequency developers perform code review also play a role in it. In line with previous work [9], these results raise questions on the effectiveness of the current development process, including coding and reviewing activities. To create a different approach, one may consider incorporating explicit security aspects in development activities, such as checklists for code review. The use of code checklists to support developers has been the object of extensive investigation [56], [57]. Studies can be designed and carried out to determine how to develop security-oriented checklists that do not overburden the reviewers, yet are effective.

Practice makes perfect – with a mentor. Participants reported how their experience with security issues (or lack thereof) played a key role in detecting (or missing) the **IIV** in the experiment. This raises the question: how can inexperienced developers be trained to find security issues? Our hypothesis is that code review might serve this purpose well. In fact, previous studies [45], [58], [59] reported how practical knowledge transfer is one of the main outcomes of the code review process. Therefore, through code review, junior developers can be guided by a more experienced developer in identifying vulnerabilities in the project code-base, with the benefit of having clear real-world examples and scenarios. Software projects can consider how to integrate this into their code review process and practices.

B. Contributions to Research and Practice

Overall, our work fits into the context of a type II [60] middle-range theory [61] as we focused on showing how and why software developers can(not) detect improper input validation vulnerabilities.

In this context, the outcomes of our study contribute with the following main points to secure software engineering research and practice:

- Our study contributes to cognitive theories of programmer errors [62], [63] and debugging [64], as well as inspection process models [65], by providing evidence on the role of explicit attack scenarios, practical knowledge and mindset, and prompting.
- Our findings motivate the need for educational research that facilitates the design and implementation of security training for developers by employing authentic [66] and experiential learning [67] techniques. For instance, our study highlights the importance of concrete attack scenarios, suggesting that vulnerabilities with not so popular scenarios should be further explored in security training.
- The observed effect of security warnings indicates how code review can be a fertile ground to use vulnerability detectors. Interdisciplinary investigations involving security as well as HCI (Human Computer Interaction) researchers can be conducted with the aim of devising ways to provide this information effectively.
- Our study supports that software professionals, particularly developers, should integrate a security-aware attitude into their practices (rather than delegating [10]) to gain the required skills while working on their code-base and avoid overlooking even simple vulnerabilities, such as **IVQI**.

VII. Conclusions

In the study we presented in this paper, we investigated to what extent developers can(not) detect *Improper Input Validation* vulnerabilities (**IIV**) and the underlying reasons. To this aim, we designed and conducted an online study that had 146 valid participants. These participants were assigned to changes with one of the following two **IIV** types: *SQL Injection* (**SQLI**) and *Improper Validation of Specified Quantity Input* (**IVQI**). The former vulnerability presents a visible, popular attack scenario.

Overall, 45% of the participants found the vulnerability. Developers were seven times more likely to detect the **SQLI**, thus confirming the role of the visible attack scenario. After warning the participants of the existence of a vulnerability in the code they just reviewed, an additional 14% of the respondents able to find the vulnerability they missed. Among the 41% of the participants who could not identify the **IIV** at all, 91% were assigned to **IVQI**.

Importantly, these results indicate a lack of knowledge and practice to identify vulnerabilities among the participants, especially when an attack scenario is not visible. The effect of the security warning provides evidence that a significant portion of developers does not focus on security by default, even during code review, but could be triggered to do so with proper team policies or adequate tooling support.

Acknowledgment

The authors would like to thank the anonymous reviewers for their thoughtful and important comments, express gratitude to the 146 valid participants 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] OWASP, "What is a vulnerability?" https://owasp.org/www-community/vulnerabilities/, 2020.

[2] R. Dhamankar, M. Dausin, M. Eisenbarth, J. King, W. Kandek, J. Ullrich, and R. Lee, "The top cyber security risks. tipping point, qualys, the internet storm center and the sans institute faculty," Tipping Point, Qualys, the Internet Storm Center and the SANS Institute faculty, Tech. Rep., 2009.

[3] "Cwe top 25 most dangerous software errors," https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html, 2020.

[4] "Owasp top ten web application security risks," https://owasp.org/www-project-top-ten/, 2020.

[5] "Gitlab: Mapping the devsecops landscape - 2020 survey," https://about.gitlab.com/developer-survey, 2020.

[6] "Cve details - list of reported vulnerabilities," https://www.cvedetails.com/vulnerability-list, 2020.

[7] K. Tsipenyuk, B. Chess, and G. McGraw, "Seven pernicious kingdoms: A taxonomy of software security errors," *Security & Privacy*, vol. 3, no. 6, pp. 81–84, 2005.

[8] "Input validation cheat sheet," https://cheatsheetseries.owasp.org/ cheatsheets/Input_Validation_Cheat_Sheet.html, 2020.

[9] I. Woon and A. Kankanhalli, "Investigation of is professionals' intention to practise secure development of applications," *International Journal of Human-Computer Studies*, vol. 65, no. 1, pp. 29 – 41, 2007.

[10] J. Xie, H. R. Lipford, and B. Chu, "Why do programmers make security errors?" in *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, 2011, pp. 161–164.

[11] J. Clarke, *SQL Injection Attacks and Defense*. Elsevier Inc., 2012.

[12] E. Galluccio, C. Edoardo, and G. Lombari, *SQL Injection Strategies*. Pakt Publishing, 2020.

[13] "xkcd: A webcomic of romance, sarcasim, math and language," https://xkcd.com/327/, 2020.

[14] A. Naiakshina, A. Danilova, C. Tiefenau, M. Herzog, S. Dechand, and M. Smith, "Why do developers get password storage wrong? A qualitative usability study," in *Proceedings of the Conference on Computer and Communications Security*, 2017, pp. 311–328.

[15] A. Naiakshina, A. Danilova, C. Tiefenau, and M. Smith, "Deception task design in developer password studies: Exploring a student sample," in *Proceedings of the Conference on Usable Privacy and Security*, 2018, pp. 297–313.

[16] A. Naiakshina, A. Danilova, E. Gerlitz, E. von Zezschwitz, and M. Smith, ""If You Want, I Can Store the Encrypted Password": A password-storage field study with freelance developers," in *Proceedings of the Conference on Human Factors in Computing Systems*, 2019, pp. 1–12.

[17] P. Morrison, T. D. Oyetoyan, and L. Williams, "Identifying security issues in software development: are keywords enough?" in *Proceedings of the International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 426–427.

[18] J. Santos, K. Tarrit, A. Sejfia, M. Mirakhorli, and M. Galster, "An empirical study of tactical vulnerabilities," *Journal of Systems and Software*, vol. 149, pp. 263–284, 2019.

[19] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *Proceedings of the International Conference on Software Engineering*, 2019, pp. 164–175.

[20] M. Mirakhorli, M. Galster, and L. Williams, "Understanding software security from design to deployment," *Software Engineering Notes*, vol. 45, no. 2, pp. 25–26, 2020.

[21] T. Scholte, D. Balzarotti, and E. Kirda, "Have things changed now? an empirical study on input validation vulnerabilities in web applications," *Computers & Security*, vol. 31, no. 3, pp. 344 – 356, 2012.

[22] M. Howard, "Mitigate security risks by minimizing the code you expose to untrusted users," https://docs.microsoft.com/en-us/archive/msdn-magazine/2004/november/security-tips-minimizing-the-code-you-expose-to-untrusted-users, 2019.

[23] A. Meneely and O. Williams, "Interactive churn metrics: Socio-technical variants of code churn," *Software Engineering Notes*, vol. 37, no. 6, pp. 1–6, 2012.

[24] Y. Shin, A. Meneely, L. Williams, and J. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *Transactions on Software Engineering*, vol. 37, pp. 772–787, 2011.

[25] A. Meneely and L. Williams, "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*, 2010, pp. 1–10.

[26] A. Meneely, A. Tejeda, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis, "An empirical investigation of socio-technical code review metrics and security vulnerabilities," in *Proceedings of the International Workshop on Social Software Engineering*, 2014, pp. 37–44.

[27] C. Thompson and D. Wagner, "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*, 2017, pp. 83–92.

[28] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: An empirical study," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2014, pp. 257–268.

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

[30] M. di Biase, M. Bruntink, and A. Bacchelli, "A security perspective on code review: The case of chromium," in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, 2016, pp. 21–30.

[31] A. L. Nichols and J. K. Maner, "The good-subject effect: Investigating participant demand characteristics," *Journal of General Psychology*, vol. 135, no. 2, pp. 151–165, 2008.

[32] "Cwe-89: Sql injection," https://cwe.mitre.org/data/definitions/89.html, 2020.

[33] "Cwe-1284: Improper validation of specified quantity in input," https://cwe.mitre.org/data/definitions/1284.html, 2020.

[34] "Cwe-20: Improper input validation," https://cwe.mitre.org/data/definitions/20.html, 2020.

[35] "Owasp," http://owasp.org/, 2020.

[36] L. Braz, E. Fregnan, G. Çalikli, and A. Bacchelli, "Data and materials for Why don't Developers Detect Improper Input Validation? `'; DROP TABLE Papers; --`," https://doi.org/10.5281/zenodo.3996696, 2021.

[37] D. Falessi, N. Juristo, C. Wohlin, B. Turhan, J. Münch, A. Jedlitschka, and M. Oivo, "Empirical software engineering experts on the use of students and professionals in experiments," *Empirical Softw. Engg.*, vol. 23, no. 1, pp. 452–489, 2018.

[38] "CRExperiment," https://github.com/ishepard/CRExperiment, 2020.

[39] D. Spadini, F. Palomba, T. Baum, S. Hanenberg, M. Bruntink, and A. Bacchelli, "Test-Driven code Review: An empirical study," in *Proceedings of the International Conference on Software Engineering*, 2019, pp. 1061–1072.

[40] D. Spadini, G. Çalikli, and A. Bacchelli, "Primers or reminders? The effects of existing review comments on code review," in *Proceedings of the International Conference on Software Engineering*, 2020, pp. 1171–1182.

[41] "Mergely," www.mergely.com, 2020.

[42] "Gerrit code review," https://www.gerritcodereview.com/, 2020.

[43] "TIOBE-Index," https://www.tiobe.com/tiobe-index/, accessed: 2020-07-15.

[44] "Programinz: Java examples and tutorials," https://www.programiz.com/java-programming/examples, 2020.

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

[46] "What to look for in a code review: Google's engineering practices documentation," https://google.github.io/eng-practices/review/reviewer/looking-for.html, 2020.

[47] S. Mcintosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Softw. Engg.*, vol. 21, no. 5, pp. 2146–2189, 2016.

[48] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

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

[50] F. Faul, E. Erfelder, A. G. Lang, and A. Buchner, "Gpower3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences," *Behavior Research Methods*, vol. 39, pp. 175 – 191, 2007.

[51] T. Baum, K. Schneider, and A. Bacchelli, "Associating working memory capacity and code change ordering with code review performance," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1762–1798, 2019.

[52] T. Cook and D. Campbell, *Quasi-Experimentation: Design and Analysis Issues for Field Settings*, 1st ed. Houghton Mifflin Company, 1979.

[53] I. MacKenzie, *Human-Computer Interaction: An Empirical Research Perspective*, 1st ed. Morgan Kaufmann Publishers Inc., 2013.

[54] G. Charness, U. Gneezy, and M. Kuhn, "Experimental methods: Between-subject and within-subject design," *Journal of Economic Behavior & Organization*, vol. 81, no. 1, pp. 1–8, 2012.

[55] R. Reinhart and G. Woodman, "High stakes trigger the use of multiple memories to enhance the control of attention," *Cerebral Cortex*, vol. 24, no. 8, pp. 2022–2035, 2014.

[56] G. Rong, J. Li, M. Xie, and T. Zheng, "The effect of checklist in code review for inexperienced students: An empirical study," in *Proceedings of theConference on Software Engineering Education and Training*, 2012, pp. 120–124.

[57] S. McConnell, *Code complete*. Pearson Education, 2004.

[58] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*, 2013, pp. 202–212.

[59] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at Google," in *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 181–190.

[60] S. Gregor, "The nature of theory in information systems," *MIS Quarterly*, vol. 30, no. 3, pp. 611–642, 2006.

[61] R. Wieringa and M. Daneva, "Six strategies for generalizing software engineering theories," *Science of Computer Programming*, vol. 101, pp. 136 – 152, 2015.

[62] I. Vessey, "Toward a theory of computer program bugs: An empirical test," *International Journal of Man-Machine Studies*, vol. 30, no. 1, pp. 23–46, 1989.

[63] A. Ko and B. Myers, "A framework and methodology for studying the causes of software errors in programming systems," *Journal of Visual Languages and Computing*, vol. 16, no. 1–2, pp. 41–84, 2005.

[64] M. Atwood and H. Ramsey, "Cognitive structures in the comprehension and memory of computer programs: an investigation of computer program debugging," Science Applications INC Englewood CO, Tech. Rep., 1978.

[65] A. Porter, H. Siy, A. Mockus, and L. Votta, "Understanding the sources of variation in software inspections," *Transactions on Software Engineering and Methodology*, vol. 7, no. 1, pp. 41–79, 1998.

[66] M. Lombardi, "Authentic learning for the 21st century: An overview," *Educause Learning Initiative*, vol. 1, no. 2007, pp. 1–12, 2007.

[67] P. Felicia, *Handbook of research on improving learning and motivation through educational games: Multidisciplinary approaches: Multidisciplinary approaches*. iGi Global, 2011.