Security Notifications in Static Analysis Tools: Developers' Attitudes, Comprehension, and Ability to Act on Them

Mohammad Tahaei
mohammad.tahaei@ed.ac.uk
School of Informatics
University of Edinburgh

Kami Vaniea
kami.vaniea@ed.ac.uk
School of Informatics
University of Edinburgh

Konstantin Beznosov
beznosov@ece.ubc.ca
Electrical and Computer Engineering
University of British Columbia

Maria Wolters
maria.wolters@ed.ac.uk
School of Informatics
University of Edinburgh

ABSTRACT

Static analysis tools (SATs) have the potential to assist developers in finding and fixing vulnerabilities in the early stages of software development requiring them to be able to understand and act on tools' notifications. To understand how helpful such SAT guidance is to developers, we ran an online experiment (N=132) where participants were shown four vulnerable code samples (SQL injection, hard-coded credentials, encryption, and logging sensitive data) along with SAT guidance, and asked to indicate the appropriate fix. Participants had a positive attitude towards both SAT notifications and particularly liked the example solutions and vulnerable code. Seeing SAT notifications also led to more detailed open-ended answers and slightly improved code correction answers. Still, most SAT (SpotBugs 67%, SonarQube 86%) and Control (96%) participants answered at least one code-correction question incorrectly. Prior software development experience, perceived vulnerability severity, and answer confidence all positively impacted answer accuracy.

CCS CONCEPTS

• **Human-centered computing → Empirical studies in HCI**;
• **Security and privacy → Usability in security and privacy**;
Software and application security.

KEYWORDS

usable security, software developers, security notifications, static analysis tools

1 INTRODUCTION

Developers write the code that goes into all types of applications world wide, but despite the existence of security support tools, like static analysis checkers, live code continues to have serious well known security vulnerabilities in it. There are several possible reasons that detectable security issues persist in deployed applications, such as not prioritising security, lack of awareness of available tools, usability issues, or the challenges around understanding and fixing identified problems. In this work, we look at Static Analysis Tools (SATs) which can scan code for known vulnerabilities and provide developers with potentially useful information about the nature of the issue and how to correct it. We specifically look at the guidance SATs provide to developers and how comprehensible and useful it is to them when correcting identified issues.

Security issues continue to impact deployed programs used by millions of users. A study of apps on the Android store found vulnerabilities in security-important programs like bank and credit card processing software [36]. Almost more concerning is how consistent vulnerabilities are over time. A glance at the OWASP top ten security risks over several years shows that the same issues that were problematic ten or more years ago remain some of the most common today [77]. So the problem is not that developers are creating new and unique issues that are challenging to detect and fix. It is that they make mistakes that are well known and have known solutions. The observation suggests that the problem is a lack of developer awareness and/or support.

SATs are designed to support developers by detecting code defects, in particular, in the early stages of software development. SATs can work without needing to run the code itself so they can be integrated into IDEs or even work off of websites which makes it easier for developers to find and fix defects earlier in the development process rather than waiting till the test and production stages. SATs are recommended as the first line of defence against vulnerabilities [102], and developers have previously acknowledged their usefulness in finding security vulnerabilities [97]. Various academic researchers recommend developers use SATs [45, 58, 86, 101, 102], and they are also used by large technology companies such as Microsoft [22, 91], Facebook [31], and Google [5, 7, 82, 83], as well as in open-source projects such as Linux, Firefox, and Qt [14, 48, 99, 105]. SATs are used for a wide range of code issues such as performance and code clarity, but developers especially value them in finding

security issues. Microsoft developers, for example, ranked finding security issues as their top reason for using a SAT [22]. Facebook also recently developed a SAT specifically to help their developers find security issues [15, 31]. SATs are also commonly used by companies that consider security to be a high priority [44].

Fixing issues is about more than just finding them. It is also about implementing a fix, which requires understanding the nature of the problem and how to correctly implement the solution. Developers come from a wide range of backgrounds, skill sets, and training [38, 65, 76]. While many of them may be aware of common issues like SQL injection, it is unreasonable to assume that all developers have a deep knowledge of all possible issues or how to fix them without support. Hence, most SATs not only locate the issue, they also provide developers with information about the identified issues and guidance on how to go about further understanding and fixing the issue. This guidance is presented in a wide variety of ways depending on the SAT, but often is presented as a set of notifications linked to problematic lines of code.

As support tools, SATs have the potential to provide developers with helpful guidance linked to their coding issues. But existing SATs suffer from usability issues such as interrupting work flow [22, 54, 88, 93], and poor integration with IDEs [54]. SAT notification content can also be hard to understand [72, 82]. Developers use web search engines to figure out what SAT notifications say [53]. At Microsoft, developers ranked "bad error messages" as their second most important issue with SATs [22], it was also the case that 75% of Google developer SAT bug reports were because of notifications misunderstandings [83]. These issues reduce the effectiveness of the tools leading to developers potentially making less use of SATs.

In this work, we conducted an online experiment to see if the notification guidance provided by two SATs (SonarQube, SpotBugs) assisted developers in identifying correct fixes to vulnerable code (SQL injection, hard-coded credentials, encryption, and sensitive data exposure) as compared to only providing the problematic line number (Control). We also asked participants follow-up comprehension and attitude questions about the provided notifications.

We found that the SpotBugs condition showed a statistically significant increase in identifying the correct fix compared to the Control. Most of the SpotBugs impact comes from the sensitive data exposure code sample, which was the only code sample of the four we tested that showed a clear difference between the three conditions and was the most challenging for developers to answer correctly. The result shows that notification content can have a positive impact on developer accuracy, but it also means that the observed improvements are not happening uniformly, even within the same tool, it also means that our choice of sample code vulnerabilities likely impacted which tool showed positive results.

We also found that prior software development experience, the perceived vulnerability severity, and confidence in answer had significant impacts on ability to find the correct fix. Developers showed a positive attitude towards notification information, particularly examples of vulnerable code and example solutions. Developers in SAT conditions also provided more detailed answers to free-text questions compared to the Control, suggesting that the notification content is impacting developer comprehension of the problem.

2 RELATED WORK

SATs work by analysing code and identifying potential known vulnerabilities in the code, often with feedback about the vulnerability and how to go about fixing it. While this process is specific to SATs, it has some obvious overlaps with areas like compiler errors and work on communicating security information to users.

2.1 Prevalence of known vulnerabilities

Well known security vulnerabilities are distressingly common in deployed code. For example, one study looked at incorrect usage of a confusing cURL PHP API call that checks if SSL certificates are valid and found that developers at large financial companies such as PayPal, Amazon, Chase mobile banking, and several other organisations had unintentionally disabled SSL verification [36]. Similarly, another study of 1,009 iOS apps shows that 14% incorrectly used SSL validations when communicating with servers [32]. Both of these vulnerabilities are fairly simplistic and easy to detect, but they also effectively disabled encryption in transit for a large number of apps and services. So while the vulnerabilities are simple and well known, the impact is large and problematic.

Vulnerabilities like those above are unfortunately easy to add into code unintentionally, particularly if the vulnerability is caused by several events stacking together. SQL injection is a good example of this issue. Data is frequently entered into SQL databases by constructing a query string. A common issue is to use unverified input in that string, essentially allowing the attacker to insert their own query. The issue is not necessarily obvious if the developer is not thinking about where their string data is coming from, but in modern APIs it can be easily fixed by using *prepared statements* which will do the input verification for the developer. A similar, more subtle, example is writing sensitive data into log files that are poorly secured. Logging events and errors is a common developer behaviour, especially when debugging, but thinking about where those logs will go long term and how they will be secured requires more awareness of issues. These types of issues can lead to serious consequences. For example in 2019, DoorDash, a food delivery app with over 10m installs, stored users' credentials in plain text on users' phones, allowing any other app access to the data [29, 30].

2.2 Static analysis tools (SATs)

SATs can assist developers in identifying security issues by scanning code without needing to run it [60]. These tools look for known patterns both in the code text as well as in the call graphs, doing so lets them find issues that may be caused by code in several different files. For example, running a static analyser on 293 open-source projects found 21,201 code "security smells" such as hard-coded credentials, weak encryption algorithms, and incorrect usage of HTTPS [80]. SATs have also been used very successfully to detect apps that potentially leak personal information through ad networks [52] and finding inconsistencies between privacy policies and implementations [106] showing their range of ability at finding problematic patterns in software.

Unfortunately, despite being effective at finding vulnerabilities, SATs have several usability problems including checking for the wrong types of problems by default, having poor notifications [6, 22, 90], interrupting work flows [22, 54, 93], having too

many false positives [22, 54], not providing enough support for teamwork, and poorly integration with IDEs [54]. Looking closer at SAT notification content, a recent heuristic walkthrough and user study of four security SATs showed that "missing or buried information" in notifications was a major usability issue. Participants used terms such as generic, complicated, unclear, or short, to describe notifications [88].

While beyond the scope of this paper, dynamic analysis is also an approach developers can use to detect vulnerabilities during code execution. Its obvious down side is that it requires that code be executed and the inputs used during execution can impact what it finds. Its advantage over SATs is primarily that it can operate on code where the full source is not available, such as with apps with third-party libraries that download and run code during execution. Both static and dynamic analysis provide useful insights for testing, but neither are comprehensive [102]. While we focus on SATs in this paper, our findings would likely apply to dynamic analysis notification content as well.

2.3 Communicating with developers

One of the most common ways developers have historically gotten feedback about their code is through compiler error messages, which provide feedback when code does not adhere to the language requirements. Given their ubiquity, developer interactions with these error messages has been studied for some years [8, 13, 85, 94]. Focusing on usability issues, we know that a developers' ability to read a compiler error message is correlated with their ability to complete a programming task correctly [9]. Error messages, however, are not necessarily easy to read and have a complexity similar to source code [9]. Prior work also suggests adding detailed explanations to compiler errors, such as concrete suggestions for common error cases, can help novice developers learn to produce fewer errors over time [12].

Several works have looked at the properties a good developer-centred notification should have. Cranor proposed a general framework for reasoning about security communication which is generally intended for end users [25], but is also applicable for developers, her framework emphasises the need for communications that can be readily understood, processed further, and remembered. Bauer et al. proposed several general principles for designing usable warning messaging including: describe the risk comprehensively, be concise and accurate, offer meaningful options, present relevant contextual information, and follow a consistent layout [11]. Gorski et al. applied Bauer's guidelines to a cryptography API design that included multiple elements such as risk description, secure and insecure actions with examples, and background information. They found that having the API provide a warning with security advice improves code security without changing the perceived usability of the API [40]. In a follow-up participatory design study with developers, researchers highlighted five key elements (message classification, title message, code location, link to detailed external resources, and colour) that participants considered helpful in cryptography API warnings [39]. On a positive note, simple changes to API messaging have been show to be helpful. In OpenSSL, for example, `hostname mismatch (X509_V_ERR_HOSTNAME_MISMATCH)` can be reworded to "`The server hostname does not match the certificate subject name. (X509_ERR_HOSTNAME_MISMATCH, see http://x509errors.cz)`", to provide more information to developers [96].

Looking at SAT notification information specifically, notification content can be confusing and require a search engine to understand the words used [53]. Empirically, developers also require an unexpectedly large amount of time to fix low-complexity SAT-identified issues [46] suggesting that comprehending and applying the notification content was not straight-forward. Google had its own developers use a SAT and found that 75% of the bugs they reported were related to misunderstandings of the notification content [83]. When Microsoft developers were asked about SAT pain points, the top two answers were: "wrong checks are on by default" and "bad error messages" [22].

These results suggest that there is room for improvement in our understanding of how developers interpret SAT notification content and how that interpretation translates into their ability to correct identified issues. We extend the body of research on developers comprehension of and attitudes towards SAT security notification [53, 54, 89, 90] by conducting a quantitative study with a larger sample and also looking at the effectiveness of SAT security notification content at assisting developers in fixing vulnerabilities.

3 METHOD

Prior research calls for designing effective security communication aimed at developers [26, 39–41, 61, 92]. We contribute to this work by looking at the effectiveness of SAT security notification content. Our research questions (RQs) are:

RQ1: how effective is SAT security notification content at assisting developers in fixing vulnerabilities?

RQ2: what are developers' attitudes toward SAT security notification content?

RQ3: how do developers' attitudes toward SAT security notification content correlate with their ability to fix vulnerabilities?

RQ4: how do developers comprehend SAT security notification content?

We conducted an online between-subjects experiment with three conditions including two SATs and one Control condition to answer our RQs. Each condition had four code samples each with one vulnerability. We showed participants (N=132) all four samples presented in a random order from one of the conditions.

3.1 Apparatus and materials

We used an online survey, deployed on Qualtrics, as it allowed us to gather data from a larger sample of developers. We tested our survey content in two rounds. First with nine colleagues who helped us improve the grammar, spelling, and clarity of the questions. Second, with a pilot of 13 Prolific participants. We used the data from both rounds to improve questions and estimate timing.

3.1.1 Tool Selection. To select which SATs to use in the study, we made a list of SATs that appeared in: (1) SAT benchmark papers [43, 79], (2) developer studies [6, 88], and (3) Stack Overflow discussions about SATs [47]. Then, we filtered for SATs that appeared in more than one source and had pre-written security rules capable of detecting a range of vulnerabilities. Our resulting list included Fortify, Veracode, Kiuwan, SonarQube, and SpotBugs. A free

version was available for SonarQube and SpotBugs. We reached out to Fortify, Veracode, and Kiuwan to give us access to their products and collaborate with us, but they either declined or did not respond to our requests (a similar scenario happened in the study by Smith et al. [88]). Hence, our final tools were SonarQube and SpotBugs.

SonarQube (v8.1.0.31237) was initially added because it appeared both in interviews with developers [99] and in Stack Overflow discussions [47]. It had an open-source community version and a paid version. We used the community version. We installed the service on a local machine (MacOS) as a web server. The tool displayed its reports through an interactive web page generated after running a command in the terminal.

SpotBugs (v3.1.13) is a successor of *FindBugs*. It has been used in recent studies [43, 79], and its predecessor appeared in several research papers for usability experiments [54, 88, 89], lessons learned from deployments [7, 82], and in Stack Overflow discussions [47]. We installed SpotBugs as a plugin in Eclipse (MacOS) through Eclipse Marketplace. We further added the *Find Security Bugs (FSB)* (v1.10.1) a plugin which allowed us to run security analysis. Reports were generated similar to other analysis that Eclipse provides.

3.1.2 Code Samples. We made a list of code samples with vulnerabilities from three resources: (1) OWASP's 2017 top ten web application security risks [77], (2) common weakness enumeration (CWE) 2019 top 25 most dangerous software errors [24], and (3) prior research in usable security studies with developers [1, 4, 28, 40, 57, 66, 71, 79]. All code samples were in Java which was selected because it was a popular programming language in several platforms such as GitHub [37], Stack Overflow [76], and other programming languages indexing services [20, 23, 51]. We adapted code samples from National Security Agency test cases [73] and Find Security Bugs code samples [18]. Our two SATs also detected different security issues; so we tested samples to make sure that both tools would produce a notification on the same code. For example, we tried several code samples with cross-site scripting vulnerabilities, but none of the samples would trigger a notification in both tools. The final cases were selected to use a range of vulnerability types:

Sensitive data exposure (DE): excessive and unnecessary logging could expose sensitive data (OWASP A3, CWE-200, and [28]). We used logging of data from a server request. If a runtime error occurs the exception would be logged using `e.printStackTrace()` possibly making the data available to an attacker with access to potentially less secured logs.

Encryption (EN): where the code encrypts plain text [1, 40, 57, 66, 71] using a key and a cipher with RC2 method. RC2 is an obsolete algorithm that can be broken [55].

Hard-coded credentials (HC): use of hard-coded credentials in the code (CWE-798 and [4, 79]) can reveal sensitive information about the software such as usernames, passwords, and API secret keys. The sample creates a database connection with hard-coded `admin` as username and password.

SQL injection (SI): data provided by users should be sanitised before use in SQL database queries (OWASP A1, CWE-89, and [28]). Our case contained an update to the database using a string concatenation including input data using the `createStatement()` and `execute()` methods.

Table 1: Notification elements that appeared in each of the SAT notifications shown to participants. N shows the number of participants in each condition (132 in total).

	Control (N=46)				SonarQube (N=43)				SpotBugs (N=43)			
	DE	EN	HC	SI	DE	EN	HC	SI	DE	EN	HC	SI
Example solution code					✓	✓	✓	✓		✓		✓
Example vulnerable code					✓	✓	✓	✓	✓	✓		✓
Explanations					✓	✓	✓	✓	✓	✓	✓	✓
Links to other resources					✓	✓	✓	✓	✓	✓		✓
Meta data (e.g. rank, severity, category)					✓	✓	✓	✓	✓	✓	✓	✓

For each code sample, SAT combination we recorded the notification text that was produced. Often this text was broken into several elements such as example solution code, links to other resources, and explanations per code sample. Table 1 shows which elements were present by SAT and code sample.

3.2 Survey instrument

Participants were first instructed to read the participant information sheet and then asked for consent. Those who consented, were tested on their proficiency in Java. They were asked to answer two multiple-choice Java questions [87]: (1) find the correct way of declaring the main method to allow a class to be started as a standalone Java program. The correct answer was `public static void main(String args[])`. (2) Identifying the keyword that is not a Java keyword from `throw`, `void`, `instanceof`, and `except`; `except` was the correct answer. If they did not get the right answer for both Java questions, they were sent to a thank you page and the survey ended. Those who answered the Java questions correctly were shown a brief introduction to the upcoming pages that directed them to use the provided notification to answer questions and not use any search engines.

For each code sample, participants were shown the vulnerable code with a highlighted line, the notification(s), and several questions. The vulnerable code was comprised of a small class with one method that included a code snippet with a vulnerability. The notification contained the exact wording and structure from the tools but was minimally re-formatted so that they used consistent fonts, colours, text width, and heading styles. Figure 1 shows the notification for HC case in the SonarQube condition. For some condition/case combinations the tool produced more than one notification, or had a short and long version. In these cases, all the notifications from the tool were shown to the participant, Figure 1 also depicts such a situation. For the Control condition, the participants were only shown the short notification: "There is a vulnerability in line [n]."

Questions for each code sample included a multiple choice question with potential fixes, three open-ended comprehension questions, and eighteen Likert items. Potential fix options were modifications of the highlighted vulnerable line with one or two neighbouring lines also shown for context, they were shown in a randomised order. The options included: unmodified copy-paste of the vulnerable sample code, correct fix to the vulnerability, and two options that performed the same insecure action using different code. The last two code options included modifications such as: (1) adding code lines with no functional impact such as adding a new variable,

The following Java code establishes a database connection. Please answer the following questions based on the code and the provided output from the static analysis checker.

```java
1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  import java.sql.SQLException;
4  import java.util.Properties;
5
6  public final Connection action() throws SQLException {
7
8      return DriverManager.getConnection("jdbc:mysql://localhost/dbName", "admin", "admin");
9  }
```

Imagine that you finish writing the method above, and then run a static analysis tool on your code. After running, the tool generates the following notification on line 8:

Remove this hard-coded password.

You ask the tool for more details and see the information below:

Credentials should not be hard-coded
- Vulnerability
- Blocker
- Main sources
- cert, cwe, owasp-a2, sans-top25-porous
- Available SinceJan 27, 2020
- SonarAnalyzer (Java)
- Constant/issue: 30min

Because it is easy to extract strings from a compiled application, credentials should never be hard-coded. Do so, and they're almost guaranteed to end up in the hands of an attacker. This is particularly true for applications that are distributed.
Credentials should be stored outside of the code in a strongly-protected encrypted configuration file or database.
It's recommended to customize the configuration of this rule with additional credential words such as "oauthToken", "secret", ...

Noncompliant Code Example
```
Connection conn = null;
try {
  conn = DriverManager.getConnection("jdbc:mysql://localhost/test?" +
      "user=steve&password=blue"); // Noncompliant
  String uname = "steve";
  String password = "blue";
  conn = DriverManager.getConnection("jdbc:mysql://localhost/test?" +
      "user=" + uname + "&password=" + password); // Noncompliant

  java.net.PasswordAuthentication pa = new java.net.PasswordAuthentication("userName", "1234".toCharArray());  // Noncompliant
```
Compliant Solution
```
Connection conn = null;
try {
  String uname = getEncryptedUser();
  String password = getEncryptedPass();
  conn = DriverManager.getConnection("jdbc:mysql://localhost/test?" +
      "user=" + uname + "&password=" + password);
```
See

- OWASP Top 10 2017 Category A2 - Broken Authentication
- MITRE, CWE-798 - Use of Hard-coded Credentials
- MITRE, CWE-259 - Use of Hard-coded Password
- CERT, MSC03-J. - Never hard code sensitive information
- SANS Top 25 - Porous Defenses
- Derived from FindSecBugs rule Hard Coded Password

Figure 1: Code sample and notification for hard-coded credentials case in SonarQube condition.

(2) performing the same action using multiple lines, such as assigning to a variable then passing the variable instead of the value, (3) passing a value directly instead of through a variable, (4) passing more or less parameters to methods calls, (5) calling a different method, and (6) changing the method signature to accept more or less parameters.

Three open-ended questions collected developers' comprehension of notification content: (1) *In a couple of sentences, explain the issue(s) highlighted by the notification(s) as if you were explaining them to a colleague.* (2) *What action(s) would you take in response to the notification?* and (3) *What could happen if you ignore the notification?* The above questions are loosely based on Wogalter's framework around information comprehension [104]. These questions were initially intended to judge comprehension accuracy, but as we discuss in results, the answers were too brief and free of context to reliably judge. Instead we use the answers to identify themes of notification content that is most salient to developers.

Each code sample also included a set of Likert attitudinal questions about the notification (see Table 9 in Appendix B for full list of Likert items). Some of the Likert items (item 2–3, 8–13, 15–18) were inspired by a previous exploratory study on SATs [53].

After working through the four code samples, participants were asked what features they found useful in the notifications. If the participant was a user of SATs, we also asked them which tools they used, for what purposes, and what they thought about SATs (Section 4.1). SAT attitudinal questions were taken from Vassallo et al.'s work [98]. Sample code questions and answer options of these questions, and Likert items, and also the answer options for SAT attitudinal questions were all randomised. The survey ended with demographics and employment status questions. All questions were

Table 2: Summary of participant recruitment per channel.

	Prolific	Github	Snowball
Invited	231	22,500	NA
Started the survey	189	161	8
Passed Java test	100	137	2
Passed Java but didn't finish the survey	39	67	1
Valid responses	61	70	1

mandatory and required an answer. The full survey instrument is included in the ACM supplementary materials.

3.3 Participants

We recruited participants through emails we collected from top 1,000 GitHub Java repositories sorted by the number of stars, Prolific [78] with screening for having knowledge of software development techniques and not being a student (resulted in 11,575 eligible participants out of 140,797 participants in total), and snowball sampling. Both Prolific and GitHub have been used to recruit in previous developer-centred security studies [3, 100]. For complete responses, participants received £9 through Prolific or an international gift card with the same amount for other methods (minimum wage in the UK was £8.72 per hour in June 2020). The study was conducted in accordance with the Ethics procedures of our institution.

Recruitment was done during June 2020. After removing paritipipants who did not pass the Java tests and did not finish the survey, we had 61 valid responses from Prolific, 70 from Github, and 1 from snowball sampling. Table 2 shows the details of recruitment channels and the number of participants that we recruited from each channel. The number of valid participants per condition were Control: 46, SonarQube: 43, and SpotBugs: 43.

It took participants a median of 44 minutes to finish the survey (mode = 32 minutes, SD = 229 minutes, and average = 107 minutes). The large standard deviation is likely caused by some participants leaving the survey and coming back later; therefore, median and mode better represent study duration.

Participants were predominantly male (88.6%), 17.4% were students in computer science (CS)-related topics, 59.1% were employed in software development, and 18.9% were employed in management, testing, security, and design related roles. The average team size for those employed in software-related jobs was 9.4 members (SD = 12 members). Table 3 shows a summary of participants' demographics, and Figure 2 shows their years of experience in software development, Java programming, and computer security.

3.4 Data analysis

For RQ1, we fitted two generalised linear models with binary score (correct / incorrect) as dependent variable. The first model was a generalised linear mixed model and consisted of the thee conditions (Control / SonarQube / SpotBugs) and years of experience as fixed effects, and participant ID as a random effect, given that we had four scores per participant in each condition [42]. The second model was a simple logistic regression and consisted of Sample Code (DE, EN, HC, SI) and years of experience. In order to determine the most appropriate metric of years of experience, we compared the fit of
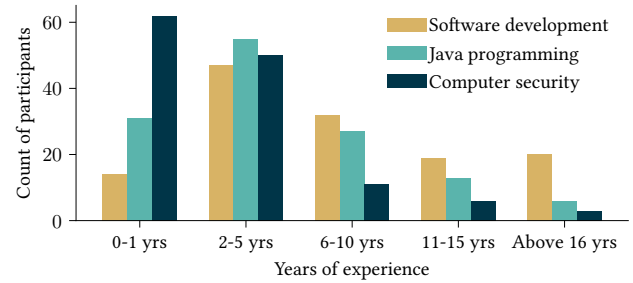
Figure 2: Participants' years of experience in software development, Java programming, and computer security (132 participants).

three models, one for years of Java experience, one for years of software development experience, and one for years of security experience resulting in years of software development experience being the best fit. Years of software experience was coded as an ordinal variable; Sample Code was a categorical variable with SI as the baseline and SAT explanation was also a categorical variable with Control as the baseline. Data was from 132 participants, each participant contributed four data points; therefore, all models were built with 528 data points (Section 4.2).

For RQ2, we conducted a factor analysis of the 18 Likert items to discover underlying factors that represent participants' ratings [16, 19] (Section 4.3). For RQ3, we used logistic regression to determine the effect of these four factors on participants' scores, while controlling for years of experience and SAT condition (Section 4.4).

All regression analyses were conducted in R using the lme4 [10] and arm [35] packages. Goodness of fit for all regression models is assessed using the Akaike Information Criterion (AIC) [34] and Nagelkerke's pseudo R^2 [67]. Regression models are reported using standard tables which list the estimate for the coefficient of each variable, together with the standard error of the estimate and the probability that it is non-zero. For easier interpretation in the text, we convert the coefficients to Odds Ratios (OR) [34]. In the context of the models, positive ORs mean that the odds of the fix being correct increase compared to the baseline by $OR - 1\%$ if the variable is 1 (e.g., if a person has 2–5 years' experience in software development) or if the unit of the variable increases by one (i.e., if the value of the attitude to notifications factor increases by one). Negative ORs mean that the odds of the fix being correct decreases correspondingly. For more on the interpretation of ORs, see [27].

In addition, RQ4 was investigated with three open-ended comprehension questions (Section 4.5). We used thematic analysis with affinity diagrams [17, 59] to identify answer themes. We chose to not classify answers into right and wrong because such judgement depends on context which was not provided (some of our participants also pointed out that answers should be context-dependent). Instead we looked at what participants were mentioning in their answers which reflects the concepts they were associating with the vulnerability and were most salient to them. Answers were at most a few sentences long which allowed us to write each down on a sticky note. We set up a board for each comprehension question (three boards in total). Then we put all sticky notes with answers

Table 3: Summary of participants demographics (N=132). Team members and number of employees are only for participants employed in computer science (CS)-related roles (N=96).

Gender		Age		Current continent		Student in CS-related topics?		Employment status		English level		Team members (N = 96)		Number of employees (N = 96)	
Male	117 (88.6%)	Mean	30.4	Europe	90 (68.2%)	Not student	109 (82.6%)	Development	78 (59.1%)	Native	57 (43.2%)	Mean	9.4	Freelancer	6 (6.2%)
Female	10 (7.6%)	SD	8	North America	23 (17.4%)	Student	23 (17.4%)	Management	9 (6.8%)	Fluent	45 (34.1%)	SD	12	2-9 employees	8 (8.3%)
Others	5 (3.8%)			Asia	12 (9.1%)			Testing	6 (4.5%)	Proficient	22 (16.7%)			10-99 employees	23 (23.9%)
				South America	4 (3%)			Security	2 (1.5%)	Conversant	5 (3.8%)			100-999 employees	24 (25%)
				Oceania	3 (2.3%)			Design	5 (3.8%)	Basic knowledge	3 (2.3%)			1,000-9,999 employees	17 (17.7%)
								Other	3 (2.3%)					10,000 or more employees	18 (18.7%)
								Employed not related to CS	16 (12.1%)						
								Not employed	13 (9.8%)						

on the boards. During virtual workshops with 5 researchers with a minimum of masters degree in CS-related topics, we asked them to read each item and then group them until all boards were stable, i.e., everyone was satisfied with the state of the board and did not want to further move the notes. We then asked them to discuss the relations between grouped notes.

3.5 Limitations

We used four code samples with two SATs, which is not representative of what developers might face in their daily routines and might not be generalisable to all developers or all possible vulnerabilities. However, the two SATs that we used in our study are amongst the most frequently used SATs for our participants (Section 4.1).

Our Control condition provided participants with the line number of the vulnerability which is more information than an unassisted developer might normally get. We chose to do so because our focus is on the content and general usefulness of the notifications in helping developers correct issues, rather than developers' ability to locate problematic lines on their own. However, this decision does mean that participants in Control condition likely performed better than they would do without any assistance.

Our population is skewed towards males with fewer years of experience (Figure 2 and Table 3), which is similar to Stack Overflow's 2020 developers survey [76]. However, the results are likely not generalisable to all developers. We distributed the survey to a sample population of developers recruited from multiple but still limited channels, which is common in developer-centred security studies. Furthermore, we did not observe participants during the survey; therefore, we cannot be sure how they completed the tasks, how long they spent on each task, or whether they consulted with any other materials and resources. It is further notable that we provided the code in the answer options (similar to what developers do when looking at sample codes in documentation or online websites such as Stack Overflow), and we did not ask participants to write the code from scratch which may lead to different findings.

The two main recruitment channels GitHub and Prolific had different variances for the rate of accuracy in code sample questions. 73.2% of GitHub participants and 54.1% of Prolific participants answered at least one code sample questions correctly. However, they were evenly distributed across SAT conditions. So while GitHub participants were more accurate, the same percentage of, say, Spot-Bugs participants were from GitHub as the other conditions. The developers who write code for millions of peoples' devices come from a wide range of backgrounds and are not all professionals. So

we believe that including both groups better represents the wider range of developers.

4 RESULTS

We present our results in four subsections. First, we present descriptive statistics of SAT usage questions to give context about our participants. Then, we move on to quantitative results (RQ1, RQ2, RQ3), followed by qualitative analysis of comprehension-related questions (RQ4).

4.1 Usage of static analysis tools

Of the 112 (85%) participants who said that they had used SATs previously, 71 (63%) use them regularly (daily, weekly, or monthly). Only one participant had never heard of SATs before.

ESLint (a JavaScript specific SAT) is used by our participants the most, which makes sense since it is commonly used in open-source projects [98]. SpotBugs + FindBugs are a close second, Checkstyle (primarily for checking coding standards) ranks third, and Sonar-Qube ranks fourth. Other tools that we considered for our study such as Veracode, Kiuwan, and Fortify are used less frequently. Participants mostly use SATs while they code (%63) (similar to the preference of Microsoft developers who prefer to see the results in their editor [22]), and during continuous integration builds (%54). Checking for style, best practices and security issues are among the top uses of SATs, which is in line with prior research [22, 98]. Participants mostly disagreed with "I do not need static analysis tools" and agreed with items that are in favour of SATs, such as they ease manual activities and SAT notifications are relevant for improving software security. Further details are included in Appendix A.

4.2 How effective is SAT security notification content at assisting developers in fixing vulnerabilities?

We used AIC to determine which of the three experience variables yielded the best fit to the data, with lower AIC indicating a better fit. The AIC of the model based on general software experience was 672.0, followed by the model based on Java experience (686.6) and the model based on security experience (689.0). We report the coefficients for the fixed effects of the resulting generalised linear mixed model in Table 4. There was only one random effect, an additional intercept that varied by participant. Typically, random effects are reported using the variance of the relevant estimates [34]. The variance of the random effect in this model was 0.01418. We see that SpotBugs improves participants' ability to find the correct fix

Table 4: Generalised linear mixed model for SAT conditions–fixed effects. Baseline is Control condition and 0–1 years experience in software development. β: Coefficient estimate. SE: standard error of coefficient estimate. Nagelkerke's pseudo R^2 = .07.

| Independent variables	β	SE	p–value
Condition			
Control		Baseline	
SonarQube	0.254	0.227	$p > .26$
SpotBugs	0.612	0.236	$p < .01$
Years of experience in software development			
0–1 years		Baseline	
2–5 years	0.873	0.323	$p < .01$
6–10 years	1.378	0.348	$p < .0001$
11–15 years	1.396	0.384	$p < .0005$
16 years and above	1.437	0.382	$p < .0005$
Intercept	-0.723	0.3001	$p < .05$

compared to the Control condition (β = 0.612, OR = 1.84, $p < .01$), but the findings for SonarQube are inconclusive.

However, the strongest effect is years of experience in developing software as can be seen from the size of the coefficients in Table 4. Participants with 2–5 years' experience perform better than those with 0–1 years of experience (β = 0.873, OR = 2.39, $p < .01$). Those with six or more years' experience perform better still. As Table 4 shows, for each of the relevant categories (6–10 years' experience, 11-15 years' experience, 16+ years' experience), the estimate of the coefficient β is between 1.38 and 1.44, which corresponds to odds ratios of around 4. Thus, experienced software developers are around four times more likely to identify the correct fix than inexperienced ones.

Only seven participants did not identify any of the correct fixes, 22 participants identified all the correct fixes, and the majority of participants (N=110) failed to identify at least one of the correct fixes. Figure 3 shows the count of correct answers for the code sample question for all participants.

Even though both tools provide vulnerable code examples, participants did somewhat better in the SpotBugs condition. Figure 4 suggests that this is mostly due to DE, where SpotBugs provided a clear advantage over the other two conditions. We hypothesise that the example solution code in SonarQube which included a logger and stack trace confused some participants; because the example's stack trace suggestion (including `LOGGER.log("context", e)` in the catch) did not appear in the catch part of the try/catch block in any of the answer options.

Overall, participants struggled most with the sensitive data exposure (DE) code sample (Figure 4). Table 5 shows the coefficients of the logistic regression model for the effect of Sample Code on the ability to find the correct fix taking the SI as the baseline. Again, we
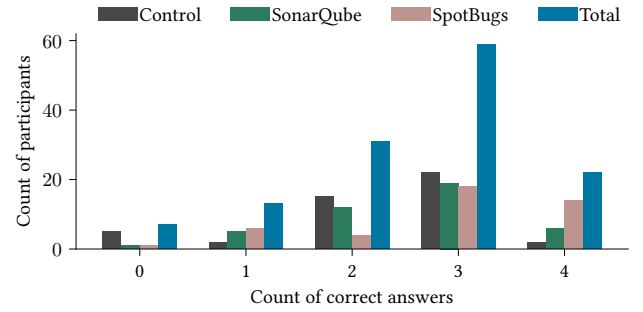
Figure 3: Number of answers per-participant that identified the correct fix to the code vulnerabilities (132 participants).
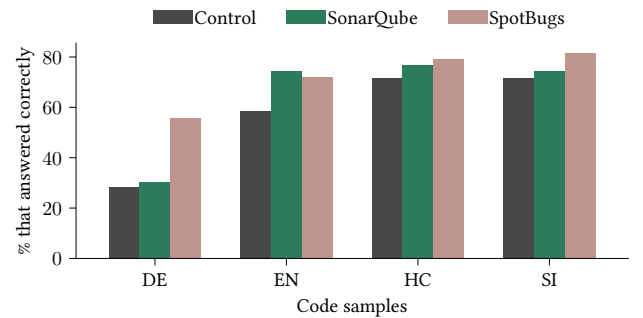
Figure 4: Percentage of participants who identified the correct code fix per code sample (132 participants).

used software experience as the relevant experience metric (AIC of model: 621.08), as it provided a better fit than Java experience (AIC: 641.49) or computer security (AIC: 642.51). The only difficult code sample is DE (β = −1.730, OR = 0.18, $p < .0001$), while HC is very similar to the baseline SI, and EN is not significantly more difficult than the baseline. This means that for every five people who manage to identify the correct fix for the baseline, SI, only one person will be able to identify the correct fix for DE. This is in line with the pattern shown in Figure 4.

## 4.3 What are developers' attitudes toward SAT security notification content?

When we asked participants to rate which elements in the presented notifications they found useful, example snippets (both of solution and vulnerable code) were rated most useful, followed by explanations. Opinions about links to other resources and metadata were mixed. Figure 5 shows combined Likert items about the usefulness of notifications elements in both SonarQube and SpotBugs.

Participants answered a set of eighteen Likert questions per code sample. To make the data easier to interpret, we used factor analysis to group items into relevant factors. Bartlett's Test of Sphericity ($p < .001$) and Kaiser-Meyer-Olkin (.87, above .50 is considered as suitable) both showed that our data is suitable for exploratory factor analysis (Cronbach's alpha = .85) [103]. We chose the number

Table 5: Logistic regression model for sample codes. Baseline is SI condition and 0–1 years experience in software development. β: Coefficient estimate. SE: standard error of coefficient estimate. Nagelkerke's pseudo R^2 = .20.

Independent variables	β	SE	p–value
Sample code			
SQL injection (SI)		Baseline	
Sensitive data exposure (DE)	-1.730	0.281	$p < .0001$
Encryption (EN)	-0.401	0.285	$p = 1.00$
Hard-coded credentials (HC)	0.000	0.296	$p > .15$
Years of experience in software development			
0–1 years		Baseline	
2–5 years	1.077	0.331	$p < .001$
6–10 years	1.625	0.359	$p < .0001$
11–15 years	1.579	0.397	$p < .0001$
16 years and above	1.660	0.395	$p < .0001$
Intercept	-0.060	0.335	$p > .85$
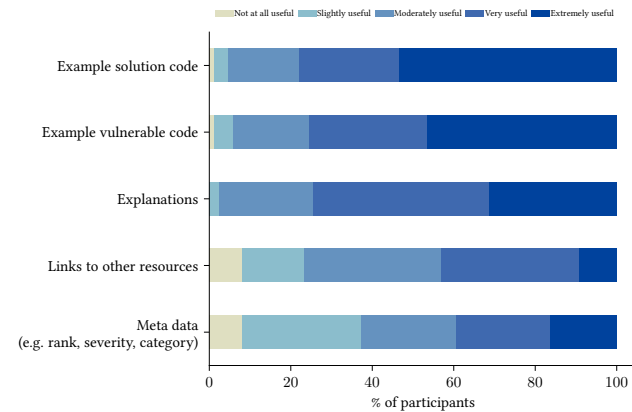
Figure 5: Answers to "to what extent did you find the following elements in the notifications useful?" From 86 SAT participants, excluding Control.

of factors by using a Scree plot and selecting eigenvalues above one [56]. Since four values were above one, we chose four factors. We tried both orthogonal and oblique rotations. We picked varimax (orthogonal rotation) because its results were slightly more interpretable. We only included items with a factor loading greater than .40 [84], which resulted in two items (17 and 18) not fitting in any factors. Table 6 shows the results of factor analysis with mean and standard deviation of each factor. It is notable that by taking the mean across the four sample codes, we effectively ignore the impact of different sample code since we are interested in the effect of conditions (tools) on the outcome variable (participants ability in finding the correct fix). Full list of Likert items with their

Table 6: Results of factor analysis for the Likert items. Three columns with conditions represent mean (μ) and standard deviation (σ) per factor taken across the Likert items in the respective factor (see Table 9 in the appendix for full list of Likert items).

	Factor	Cronbach's alpha	Control		SonarQube		SpotBugs	
			μ	σ	μ	σ	μ	σ
1	Attitudes towards the notifications	.87	2.85	0.35	3.87	0.22	3.73	0.27
2	Prior knowledge and experience	.78	3.68	0.22	3.41	0.28	3.44	0.25
3	Importance and severity of the vulnerability	.69	4.02	0.24	4.22	0.27	4.22	0.21
4	Confidence in the solution	.79	3.54	0.17	3.80	0.13	3.92	0.10

loading factors, means, and standard deviations are included in the Appendix B.

Looking at the first factor in Table 6, *attitudes towards the notifications*, the means of the two SATs are not much different, however, the differences between the SATs and the Control condition is stronger suggesting that participants have a positive attitude towards the notification compared to a minimum notification condition (Control). This is inline with Figure 5 in which the majority of participants find example code and explanations generally useful.

4.4 How do developers' attitudes toward SAT security notification content correlate with their ability to fix vulnerabilities?

Table 7 shows the effect of developer attitudes, years of software experience, and SAT condition on participants' scores. While the effect of years of experience remains unchanged , the effect of SAT condition is no longer significant. Instead, we find that participants' attitudes are much better predictors of their score. If they deem the vulnerability sufficiently important ($\beta = 0.741, OR = 2.1, p < .0001$), and are confident in their solution ($\beta = 0.508, OR = 1.67, p < .0005$), they are more likely to be correct, regardless of the SAT condition. In order to assess whether the SAT condition mediated the effect of developer attitudes on their scores, we fitted a second model that contained an interaction between Condition and Importance of Vulnerability and an interaction between Condition and Confidence in Solution. The AIC of the more complex model was 613.29; the AIC of the original model was 605.48, which means that the original model is the best fit.

4.5 How do developers comprehend SAT security notification content?

Here, we discuss the three open-ended comprehension questions which asked about explaining the issue(s) to a colleague, what actions to take in response, and what would happen if the notification was ignored. We combined the answers from participants that saw either of the two SAT notifications because we were more interested in how having notification text impacted answers. Since answers were rather short (on average 16 words, SD = 12 words) and addressed the question directly themes naturally formed around the questions. Table 8 shows the results of the thematic analysis for these questions. 29 SAT and 6 Control answers were excluded in the analysis because they were too short or had too little context to make sense of. Participants' quotes are labelled as Control (CN),

Table 7: Logistic regression model for participants' attitudes. Baseline is Control condition and 0–1 years experience in software development. β: Coefficient estimate. SE: standard error of coefficient estimate. Nagelkerke's pseudo R^2 = .24.

Independent variables	β	SE	p–value
Factor			
Attitudes to notifications	0.214	0.168	$p > .25$
Prior knowledge and experience	-0.267	0.131	$p < .05$
Importance and severity of vulnerability	0.741	0.148	$p < .0001$
Confidence in solution	0.508	0.136	$p < .0005$
Condition			
Control		Baseline	
SonarQube	-0.251	0.286	$p > .4$
SpotBugs	0.140	0.23	$p > .6$
Years of experience in software development			
0–1 years		Baseline	
2–5 years	0.800	0.344	$p < .05$
6–10 years	1.60	0.382	$p < .0001$
11–15 years	1.39	0.417	$p < .001$
16 years and above	1.80	0.445	$p < .0001$
Intercept	-5.183	0.755	$p < .0001$

SonarQube (SQ), and SpotBugs (SB) along with the participant's identification number.

While we were keen to understand participants' comprehension, the short answers made it difficult to effectively categorise answers based on comprehension level. For example, SB111's answer: *"That type of Cipher is not secure enough"* where they clearly understood that the cipher was the issue but it is unclear if they understood that they needed to use AES/GCM/NoPadding. Consequently, we chose to focus the themes brought up by participants through thematic analysis rather than a judgement of if they did or did not fully comprehend the notification content. Doing so allowed us to understand what the key aspects of the content participants found most salient and were most able to express.

4.5.1 Sensitive data exposure (DE). Participants explained DE as generic sensitive data exposure, credentials exposure, program structure and internals exposure, and file and path exposure. Interestingly, several SAT participants (N=28) did not mention a security issue in their explanations. *"Catching the exception is good but logging the exact exception gives a benefit in tracking down the code block that raised an exception"* (SQ13). In the Control condition, many participants used generic language like "it's not secure" instead of terms such as *logger* (only one CN participant mentioned a logger) and *sensitive data exposure* or *leak*, which were common in SAT conditions.

Use of a logger was the dominant response for SATs conditions, which is expected as in the SonarQube notification explicitly mentioned a logger. *"Use a logging library; add a logger to the class"* (SQ118). Another major group of participants (N=20) were uncertain about what actions to take, nine of which were not able to find the correct fix. Again, we observed that some SAT participants

(N=10) did not mention any security-related terms. Similar to the explanations, "logger" was used rarely by Control participants.

When it comes to the consequences of ignoring this notification, several SAT participants (N=27) brought up non-security related consequences such as debugging problems: *"Ignoring this notification would make the search for errors less practical"* (SQ23).

4.5.2 Encryption (EN). Most participants explained the issue as an encryption problem (outdated or weak encryption and cipher), some used data integrity which could be a result of SpotBugs's title message including data integrity: *"Data integrity is at risk, and hashing should be used to avoid such attacks"* (SB86). Four participants indicated that they did not understand the issue: *"I'm unable to explain correctly this issue"* (SB97), however, they all found the correct fix. It is interesting to see that eleven Control participants did not have a concrete explanation (out of which six were able to find the correct fix and five were not), which shows that notifications may have assisted participants in building a structured explanation, and perhaps in finding the correct fix.

Twenty participants in the SAT conditions were uncertain about what action to take or did not provide a clear action; out of this group seven were not able to find the correct fix. For example, SB37 explained: *"I would have said to use private and public keys (I only know the theory, not how to code this)."* However, several participants provided examples, or wrote that they would change the encryption to a stronger one such as AES. HMAC which appears in SpotBugs example code was not present in the responses of Control participants at all, however, AES was mentioned (N=5) suggesting awareness of it.

In regards to consequences, data tampering and not being secure were most common. A few participants brought up the idea of trade-offs. Specifically, SQ74 suggested: *"This depends on the use case. Some use case does not require strong cipher"*. While only one SAT participant saw a non-security issue with this code sample, five participants in the Control condition brought up non-security consequences of ignoring the notification: *"While running the code on different platform it could give different results or not run at all"* (CN25). Out of the two SAT participants who said nothing would happen, one was not able to find the correct fix.

4.5.3 Hard-coded credentials (HC). Many SAT participants (N=55) explained HC as credentials being visible. Some SAT participants (N=14) provided more detailed explanations such as the use of external files, encryption, or environment variables: *"The username and password should be called from the db encrypted in order to avoid middleman attacks"* (SQ30). Encryption-related terms were only used by SAT participants, which might be because of SonarQube's notification, where it recommends using an encrypted configuration file for credentials. It is further notable that vague explanations like "some security issue" only happened in the Control. SAT participants all provided detailed explanations, e.g. *"The code associated with entering the database (and its security) is probably too simple and easy to crack"* (CN19).

Suggested actions for this code sample included moving credentials from the code to configuration files or databases, and retrieving them just-in-time with function calls: e.g., *"Move account and password data to the configuration file"* (SQ132). Interestingly, six SAT

Table 8: Resulting groups from thematic analysis of three open-ended comprehension questions. SATs columns show the combined number of sticky notes in both SAT conditions, and CN shows the number of sticky notes in the Control condition per group.

Sensitive data exposure (DE)			Encryption (EN)			Hard-coded credentials (HC)			SQL injection (SI)		
	SATs	CN		SATs	CN		SATs	CN		SATs	CN
In a couple of sentences, explain the issue(s) highlighted by the notification(s) as if you were explaining them to a colleague.											
Information leak through prints & logs			Encryption - outdated methods	17	4	Password is visible	55	27	Formatting & parameters	26	1
Generic sensitive data exposure	36	7	Manipulation of data (preventing manipulation)	17		Advice, problem-scoping, teaching			String concatenation not safe	20	1
Credentials	5		Encryption - used is weak	15	12	Use external files, config files	7	1	Use prepared statements	14	1
Program structure & internals	5	4	Inappropriate use of cipher (insecure cipher)	12	14	Use encryption	5		SQL injection with no concrete explanation	7	19
Files & paths	4		Weak encryption - alternative suggested	10	4	Use environment variables	2		Sanitise input	5	
Use a logger			Data integrity	8		Visible in decompiles	8	3	Escape input	4	3
Security not mentioned	12	1	Don't know	4	1	Security not mentioned	7	4	Includes an example SQL attack	4	
Exposure mentioned	4		Laws and regulations	1		Some security issues could happen		9	Non-security issues	2	14
No security issues mentioned	16	16	Uncertain, no concrete explanation		11				Some security issue		6
Not sure		3									
It's not secure		14									
What action(s) would you take in response to the notification?											
Use a logger	27	6	Uncertain, no concrete action	20	21	Remove from code, store in a separate location	21	9	Use prepared statements	31	17
Uncertain, no concrete action	20	16	Change & use stronger example	14	7	Secure, encrypt	17	4	Uncertain, no concrete action	18	17
Use a custom error msg	11	5	Change encryption	10	7	Use a config file	13	12	Use parameters	10	4
Other non-security related	10	10	Use HMAC for encryption	10		Don't use plain password with no clear solution	9	2	Use escaping or sanitisation	9	4
Log somewhere else	7	4	Change to AES (with a concrete example)	9	5	Use function calls	7	1	Use string concatenation	6	1
Hide output	4	4	Change cipher	8	4	Uncertain, no concrete action	6	9	Other solutions	6	2
Avoid stack trace printing	3		Change & use to a recommended one	4		Use a database	4	1	Fix SQL, no clear solution given	3	1
			Other solutions	4	2	Use environment variables	4	6			
			Change to AES (with a concrete example) as recommended	3		Other solutions	4	2			
What could happen if you ignore the notification(s)?											
Sensitive info exposure & leak	27	13	Data tampering, compromise	30	3	Attacker gets access	34	18	Database exploit	30	9
Attacks could happen (access)	26	15	Not secure	19	25	Credentials exposed	22	12	Database damage (SQL injection)	27	17
No security consequences	23	10	Could be decrypted and attacked	17	6	Not secure	6	7	Unsafe database	16	5
Some security issues could happen	8	5	Could be decrypted	13	6	Other consequences	5	3	Some security issues	10	3
Debugging problems (non-security related)	5	1	It depends	3		Non-security issues	4	6	Non-security issues	6	10
Unsure		2	Nothing, uncertain	2	1	Nothing, uncertain, or a trade off	3		Don't know		1
			Consequences to other parties	2							
			Non-security issues	1	5						

participants did not provide a concrete action, five of this group were not able to find the correct fix either.

Common consequences were the exposure of credentials and attackers gaining access: *"Database could be compromised by an attacker who has visibility of the source code"* (SQ83). While three participants provided little information about the consequences, or said it is a trade-off, all the three were able to find the correct fixes.

4.5.4 SQL injection (SI). A majority of SAT participants (N=60) used terms presented by both tools such as concatenation, formatting, and prepared statements in their explanations, e.g., *"The string concatenation is not safe to use"* (SQ15). Control participants used a vague language such as "SQL injection" and "some security issues could happen." Fourteen Control participants did not bring up a security issue in their explanation, which is a large number compared to the two SAT participants (N=2). For example, CN42 explained that *"Java will not insert data into the database table."*

While several SAT participants (N=31) said that they would use PreparedStatement to fix the code, eighteen participants were not clear about what action they would take (nine out of these were not able to find the correct fix), e.g., *"Modify the code to make it safer"* (SQ51). Common consequences of SI were an "unsafe database" or "leads to exploiting" or "damaging the database." To illustrate, SQ23 explained: *"Ignoring the notification puts the database at risk if queries are coming from untrusted sources."*

With our qualitative analysis, we are not able to state whether the answers to the comprehension questions directly impacted participant's ability to identify the correct fix. We observe that a portion of participants in cases with uncertain answers were

not able to find the correct fix, however, further investigation potentially with an interview study in a lab environment where the researchers can observe participants and ask questions is needed to understand the relation between comprehension and ability to act on the notifications.

5 DISCUSSION AND FUTURE WORK

Prior work shows that if developers are prompted, nudged, or asked explicitly about security, they are more likely to choose secure solutions over insecure solutions [50, 68–70, 74]. SATs have the potential to promote more secure coding by proactively identifying issues at early stages of development along with identifying the problematic line(s) and providing specific guidance on how to correct them. The vulnerabilities we tested were fairly common in that they are well known and yet still appear in live code, even for simple errors like leaving in hard-coded credentials [21, 49, 75, 79, 80]. Yet, for all code samples, at least 20% of participants indicated an incorrect solution. Out of 132 participants, only 22 found the correct fix for all code samples, and 110 participants failed to find the correct fix for at least one of the code samples (Section 4.2). These results suggest that, at least for our code samples, the existing notification content is not sufficiently helping developers.

False positives are a major pain point for developers [22, 54] resulting in work to improve detection accuracy [81, 95]. But even if the accuracy goes up, when developers receive notification there is a real chance that they may not able to fix the vulnerability due to not understanding the issue, opening a future research directions for improving the notifications effectiveness.

One possible explanation of the lack of impact from notifications is that notification elements like presentation, phrasing, organisation, and structure are not well aligned with developer needs. Prior work on the usability of developer-aimed security communications has similarly observed that participants do not find all the information presented in notifications equally useful [39, 88]. The results are similar to our observations that participants did not find links to external resources and meta data such as rank and category as useful as sample code (Section 4.3). Participants' positive attitude towards sample solutions and vulnerable code suggests that enhancing these elements might improve usability.

Another possible explanation lies in the impact of experience and developer attitudes on their performance. It appears that more seasoned developers may benefit less from SATs, in particular if they deem the vulnerability to be important, and if they are confident in their own judgement.

Research has already suggested that notifications can be improved by including examples [62–64]. Improving examples in cryptography APIs is associated with strong improvement in participants' ability to write secure code [64]. Interestingly, most of the notifications we tested did contain example code (Table 1) but we observed effectively no impact from its inclusion or exclusion. DE and HC had no example code in SpotBugs but did in SonarQube. Yet, SpotBugs participants answered correctly more often for both code samples (Section 4.2). This outcome suggests that how examples are chosen and how well they align with the problem the developer is facing may strongly impact their usefulness. One limitation of our work is that we provided developers with a list of potential solutions, which in some cases differed from the example solutions because the solution did not match the sample code case. While a limitation, the situation also highlights a realistic event where provided solutions do not directly match the problem.

SATs are also valuable to developers because the code they are using may not always be code they themselves have written. Developers tend to use code from online resources such as Stack Overflow that may not be secure [2, 33]. They also sometimes inherit code or join projects mid-way through. Using SATs could help developers in deciding what code snippets are safe to use or and providing guidance on how to make them more secure.

Our developer population was drawn from GitHub and Prolific which are different sample sources and indeed they had different profiles. GitHub users, for example, were selected because they were part of existing Java projects. Yet participants from both pools still struggled to correctly indicate the code sample fix from among provided options (Section 3.5). We argue that developers from multiple skill levels likely need the guidance that SATs are intended to provide. It is fairly easy to fall into a trap of assuming that senior software engineers know everything, it is equally easy to assume that novice ones, or those just learning to code have serious knowledge gaps. But our work suggests that improving SAT notifications may possibly help people at all levels. Particularly if tools, like SATs, can adjust to the context and needs of developers [26]. There is room for future research to explore the different information and guidance needs developers have and how to best support them.

Comprehension questions showed that several participants were able to explain the issues mentioned in the notifications in a structured way, consider the right action, and determine the potential consequences of ignoring the notification. SAT participants used more concrete language with specific terms, as opposed to Control participants (Section 4.5). Another observation is that many participants brought up clear security consequences for code samples with obvious security elements such as EN, HC, and SI. These results suggest that notification content has some impact on developers' comprehension of the problem. Though not all the time, as in the DE code sample where few participants mentioned a security consequence. Future research could expand on the potential severity, common knowledge, and developer beliefs about different vulnerabilities. Finding out how pre-existing knowledge plays a role in developers decision making while fixing vulnerabilities could help designers make improvements to security notifications and how SATs represent information to developers.

6 CONCLUSION

We studied security notifications from two popular SATs (SonarQube and SpotBugs) and found that our participants were generally able to find the correct fix for four types of vulnerabilities with or without the assistance of SAT notification content. Only a short notification which points out the vulnerability's line number was sufficient for developers to identify the correct fix. Participants had positive attitudes towards notifications including vulnerable and compliant code examples. Participants' perceived comprehension of the notification also positively correlated with their ability to chose the correct fix.

ACKNOWLEDGMENTS

REFERENCES

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. 2017. Comparing the Usability of Cryptographic APIs. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 154–171. https://doi.org/10.1109/SP.2017.52

[2] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. 2016. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 289–305. https://doi.org/10.1109/SP.2016.25

[3] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L. Mazurek, and Sascha Fahl. 2017. Security Developer Studies with GitHub Users: Exploring a Convenience Sample. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*. USENIX Association, Santa Clara, CA, 81–95. https://www.usenix.org/conference/soups2017/technical-sessions/presentation/acar

[4] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. 2019. CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses. In *2019 IEEE Cybersecurity Development (SecDev)*. IEEE, 49–61. https://doi.org/10.1109/SecDev.2019.00017

[5] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (Sept. 2008), 22–29. https://doi.org/10.1109/MS.2008.130

[6] Nathaniel Ayewah and William Pugh. 2008. A Report on a Survey and Study of Static Analysis Users. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems* (Seattle, Washington) *(DEFECTS '08)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/1390817.1390819

[7] Nathaniel Ayewah and William Pugh. 2010. The Google FindBugs Fixit. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Trento, Italy) *(ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 241–252. https://doi.org/10.1145/1831708.1831738

[8] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. 2018. How Should Compilers Explain Problems to Developers?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 633–643. https://doi.org/10.1145/3236024.3236040

[9] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do Developers Read Compiler Error Messages?. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 575–585. https://doi.org/10.1109/ICSE.2017.59

[10] Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. 2015. Fitting Linear Mixed-Effects Models Using lme4. *Journal of Statistical Software, Articles* 67, 1 (2015), 1–48. https://doi.org/10.18637/jss.v067.i01

[11] Lujo Bauer, Cristian Bravo-Lillo, Lorrie Cranor, and Elli Fragkaki. 2013. *Warning Design Guidelines*. Technical Report. Carnegie Mellon University. https://www.cylab.cmu.edu/_files/pdfs/tech_reports/CMUCyLab13002.pdf

[12] Brett A. Becker. 2016. An Effective Approach to Enhancing Compiler Error Messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 126–131. https://doi.org/10.1145/2839509.2844584

[13] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, and et al. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland UK) *(ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 177–210. https://doi.org/10.1145/3344429.3372508

[14] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 470–481. https://doi.org/10.1109/SANER.2016.105

[15] Graham Bleaney and Sinan Cepel. 2020. Pysa: An open source static analysis tool to detect and prevent security issues in Python code. Retrieved August 2020 from https://engineering.fb.com/security/pysa/

[16] Harry N Boone and Deborah A Boone. 2012. Analyzing Likert Data. *Journal of Extension* 50, 2 (2012), 1–5. https://www.joe.org/joe/2012april/pdf/JOE_v50_2tt2.pdf

[17] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. https://doi.org/10.1191/1478088706qp063oa

[18] Find Security Bugs. 2020. Find Security Bugs. Retrieved June 2020 from https://github.com/find-sec-bugs/find-sec-bugs/tree/master/findsecbugs-samples-java

[19] James Carifio and Rocco Perla. 2008. Resolving the 50-year debate around using and misusing Likert scales. *Medical Education* 42, 12 (Dec. 2008), 1150–1152. https://doi.org/10.1111/j.1365-2923.2008.03172.x

[20] Stephen Cass. 2020. Top Programming Languages 2020. Retrieved July 2020 from https://spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020

[21] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. 2020. An Empirical Assessment of Security Risks of Global Android Banking Apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1310–1322. https://doi.org/10.1145/3377811.3380417

[22] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[23] TIOBE The Software Quality Company. 2020. TIOBE Index. Retrieved June 2020 from https://www.tiobe.com/tiobe-index/

[24] The MITRE Corporation. 2020. Common Weakness Enumeration (CWE) Top 25 Most Dangerous Software Errors. Retrieved August 2020 from https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html

[25] Lorrie Faith Cranor. 2008. A Framework for Reasoning about the Human in the Loop. In *Proceedings of the 1st Conference on Usability, Psychology, and Security* (San Francisco, California) *(UPSEC'08)*. USENIX Association, USA, Article 1, 15 pages. https://www.usenix.org/legacy/events/upsec08/tech/full_papers/cranor/cranor.pdf

[26] Anastasia Danilova, Alena Naiakshina, and Matthew Smith. 2020. One Size Does Not Fit All: A Grounded Theory and Online Survey Study of Developer

[27] H T Davies, I K Crombie, and M Tavakoli. 1998. When can odds ratios mislead? *BMJ (Clinical research ed.)* 316, 7136 (March 1998), 989–991. https://doi.org/10.1136/bmj.316.7136.989 Publisher: British Medical Journal.

[28] Pieter De Cremer, Nathan Desmet, Matias Madou, and Bjorn De Sutter. 2020. Sensei: Enforcing secure coding guidelines in the integrated development environment. *Software: Practice and Experience* 50, 9 (2020), 1682–1718. https://doi.org/10.1002/spe.2844

[29] CVE Details. 2019. CVE-2019-17397 : In the DoorDash application through 11.5.2 for Android, the username and password are stored in the log during authentication, and may be available to attackers via logcat. Retrieved June 2020 from https://www.cvedetails.com/cve/CVE-2019-17397/

[30] CVE Details. 2020. CVE security vulnerabilities related to CWE (Common Weakness Enumeration) 532. Retrieved June 2020 from https://www.cvedetails.com/vulnerability-list/cweid-532/vulnerabilities.html

[31] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. https://doi.org/10.1145/3338112

[32] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. 2013. Rethinking SSL Development in an Appified World. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/2508859.2516655

[33] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack Overflow Considered Harmful? The Impact of Copy Paste on Android Application Security. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 121–136. https://doi.org/10.1109/SP.2017.31

[34] Andrew Gelman and Jennifer Hill. 2007. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge University Press, Cambridge, UK.

[35] Andrew Gelman, Masanao Yajima Yu-Sung Su, Jennifer Hill, Maria Grazia Pittau, Jouni Kerman, Tian Zheng, and Vincent Dorie. 2020. arm: Data Analysis Using Regression and Multilevel/Hierarchical Models. Retrieved December 2020 from https://cran.r-project.org/package=arm

[36] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 38–49. https://doi.org/10.1145/2382196.2382204

[37] GitHub. 2019. The State of the Octoverse. Retrieved August 2020 from https://octoverse.github.com/

[38] Google. 2020. Google Diversity Annual Report. Retrieved August 2020 from https://diversity.google/annual-report/

[39] Peter Leo Gorski, Yasemin Acar, Luigi Lo Iacono, and Sascha Fahl. 2020. Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376142

[40] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl. 2018. Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*. USENIX Association, Baltimore, MD, 265–281. https://www.usenix.org/conference/soups2018/presentation/gorski

[41] Matthew Green and Matthew Smith. 2016. Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security & Privacy* 14, 05 (Sept. 2016), 40–46. https://doi.org/10.1109/MSP.2016.111

[42] UCLA: Statistical Consulting Group. 2020. Mixed Effects Logistic Regression. Retrieved December 2020 from https://stats.idre.ucla.edu/stata/dae/mixed-effects-logistic-regression/

[43] Andrew Habib and Michael Pradel. 2018. How Many of All Bugs Do We Find? A Study of Static Bug Detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 317–328. https://doi.org/10.1145/3238147.3238213

[44] Julie M. Haney, Mary Theofanos, Yasemin Acar, and Sandra Spickard Prettyman. 2018. "We make it a big deal in the company": Security Mindsets in Organizations that Develop Cryptographic Products. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*. USENIX Association, Baltimore, MD, 357–373. https://www.usenix.org/conference/soups2018/presentation/haney-mindsets

[45] Michael Huth and Flemming Nielson. 2019. Static Analysis for Proactive Security. In *Computing and Software Science: State of the Art and Perspectives*. Springer International Publishing, Cham, 374–392. https://doi.org/10.1007/978-3-319-91908-9_19

[26] Preferences for Security Warning Types. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, 13 pages. https://doi.org/10.1145/3377811.3380387

[46] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. 2019. How Do Developers Act on Static Analysis Alerts? An Empirical Study of Coverity Usage. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 323–333. https://doi.org/10.1109/ISSRE.2019.00040

[47] Nasif Imtiaz, Akond Rahman, Effat Farhana, and Laurie Williams. 2019. Challenges with Responding to Static Analysis Tool Alerts. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 245–249. https://doi.org/10.1109/MSR.2019.00049

[48] Nasif Imtiaz and Laurie Williams. 2019. A Synopsis of Static Analysis Alerts on Open Source Software. In *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security* (Nashville, Tennessee, USA) *(HotSoS '19)*. Association for Computing Machinery, New York, NY, USA, Article 12, 3 pages. https://doi.org/10.1145/3314058.3317295

[49] Mazharul Islam, Sazzadur Rahaman, Na Meng, Behnaz Hassanshahi, Padmanabhan Krishnan, and Danfeng (Daphne) Yao. 2020. Coding Practices and Recommendations of Spring Security for Enterprise Applications. In *2020 IEEE Secure Development (SecDev)*. IEEE, 49–57. https://doi.org/10.1109/SecDev45635.2020.00024

[50] Shubham Jain, Janne Lindqvist, et al. 2014. Should I protect you? Understanding developers' behavior to privacy-preserving APIs. In *Workshop on Usable Security (USEC'14)*. Internet Society, 10 pages. https://doi.org/10.14722/usec.2014.23045

[51] JetBrains. 2020. The State of Developer Ecosystem. Retrieved August 2020 from https://www.jetbrains.com/lp/devecosystem-2020/

[52] Ling Jin, Boyuan He, Guangyao Weng, Haitao Xu, Yan Chen, and Guanyu Guo. 2019. MAdLens: Investigating into Android In-App Ad Practice at API Granularity. *IEEE Transactions on Mobile Computing* PP (2019), 18 pages. https://doi.org/10.1109/TMC.2019.2953609

[53] Brittany Johnson, Rahul Pandita, Justin Smith, Denae Ford, Sarah Elder, Emerson Murphy-Hill, Sarah Heckman, and Caitlin Sadowski. 2016. A Cross-Tool Communication Study on Program Analysis Tool Notifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/2950290.2950304

[54] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[55] John Kelsey, Bruce Schneier, and David Wagner. 1997. Related-key cryptanalysis of 3-WAY, Biham-DES,CAST, DES-X, NewDES, RC2, and TEA. In *Information and Communications Security*, Yongfei Han, Tatsuaki Okamoto, and Sihan Qing (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 233–246. https://doi.org/10.1007/BFb0028479

[56] Jae-on Kim and Charles Mueller. 2020. *Factor Analysis*. SAGE Publications, Inc, Thousand Oaks, California. https://doi.org/10.4135/9781412984256

[57] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. 2017. CogniCrypt: Supporting Developers in Using Cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 931–936. https://doi.org/10.1109/ASE.2017.8115707

[58] James A. Kupsch, Elisa Heymann, Barton Miller, and Vamshi Basupalli. 2017. Bad and good news about using software assurance tools. *Software: Practice and Experience* 47, 1 (2017), 143–156. https://doi.org/10.1002/spe.2401

[59] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. 2017. Chapter 8 - Interviews and focus groups. In *Research Methods in Human Computer Interaction* (second edition ed.), Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser (Eds.). Morgan Kaufmann, Boston, 187–228. https://doi.org/10.1016/B978-0-12-805390-4.00008-X

[60] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67–95. https://doi.org/10.1016/j.infsof.2017.04.001

[61] Na Meng, Stefan Nagy, Danfeng (Daphne) Yao, Wenjie Zhuang, and Gustavo Arango Argoty. 2018. Secure Coding Practices in Java: Challenges and Vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 372–383. https://doi.org/10.1145/3180155.3180201

[62] Kai Mindermann, Philipp Keck, and Stefan Wagner. 2018. How Usable Are Rust Cryptography APIs?. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 143–154. https://doi.org/10.1109/qrs.2018.00028

[63] Kai Mindermann and Stefan Wagner. 2018. Usability and Security Effects of Code Examples on Crypto APIs. In *2018 16th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 1–2. https://doi.org/10.1109/PST.2018.8514203

[64] Kai Mindermann and Stefan Wagner. 2020. Fluid Intelligence Doesn't Matter! Effects of Code Examples on the Usability of Crypto APIs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 306–307. https://doi.org/10.1145/3377812.3390892

[65] Xenia Mountrouidou, David Vosen, Chadi Kari, Mohammad Q. Azhar, Sajal Bhatia, Greg Gagne, Joseph Maguire, Liviana Tudor, and Timothy T. Yuen. 2019. Securing the Human: A Review of Literature on Broadening Diversity in Cybersecurity Education. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland UK) *(ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 157–176. https://doi.org/10.1145/3344429.3372507

[66] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through Hoops: Why Do Java Developers Struggle with Cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 935–946. https://doi.org/10.1145/2884781.2884790

[67] Nico JD Nagelkerke et al. 1991. A note on a general definition of the coefficient of determination. *Biometrika* 78, 3 (09 1991), 691–692. https://doi.org/10.1093/biomet/78.3.691

[68] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, and Matthew Smith. 2020. On Conducting Security Developer Studies with CS Students: Examining a Password-Storage Study with CS Students, Freelancers, and Company Developers. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376791

[69] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, Emanuel von Zezschwitz, and Matthew Smith. 2019. "If You Want, I Can Store the Encrypted Password": A Password-Storage Field Study with Freelance Developers. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland UK) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300370

[70] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. 2017. Why Do Developers Get Password Storage Wrong? A Qualitative Usability Study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 311–328. https://doi.org/10.1145/3133956.3134082

[71] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. 2017. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 1065–1077. https://doi.org/10.1145/3133956.3133977

[72] Lisa Nguyen Quang Do, James Wright, and Karim Ali. 2020. Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations. *IEEE Transactions on Software Engineering* PP (2020), 13 pages. https://doi.org/10.1109/TSE.2020.3004525

[73] National Institute of Standards and Technology (NIST). 2017. Software Assurance Reference Dataset. Retrieved June 2020 from https://samate.nist.gov/SARD/testsuite.php

[74] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. 2014. It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots. In *Proceedings of the 30th Annual Computer Security Applications Conference* (New Orleans, Louisiana, USA) *(ACSAC '14)*. Association for Computing Machinery, New York, NY, USA, 296–305. https://doi.org/10.1145/2664243.2664254

[75] Marten Oltrogge, Erik Derr, Christian Stransky, Yasemin Acar, Sascha Fahl, Christian Rossow, Giancarlo Pellegrino, Sven Bugiel, and Michael Backes. 2018. The Rise of the Citizen Developer: Assessing the Security Impact of Online App Generators. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 634–647. https://doi.org/10.1109/SP.2018.00005

[76] Stack Overflow. 2020. Developer Survey Results. Retrieved August 2020 from https://insights.stackoverflow.com/survey/2020

[77] OWASP. 2017. The Ten Most Critical Web Application Security Risks. Retrieved August 2020 from https://owasp.org/www-project-top-ten

[78] Eyal Peer, Laura Brandimarte, Sonam Samat, and Alessandro Acquisti. 2017. Beyond the Turk: Alternative platforms for crowdsourcing behavioral research. *Journal of Experimental Social Psychology* 70 (2017), 153–163. https://doi.org/10.1016/j.jesp.2017.01.006

[79] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. 2019. CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. ACM, New York, NY, USA, 2455–2472. https://doi.org/10.1145/3319535.3345659

[80] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 164–175. https://doi.org/10.1109/ICSE.2019.00033

[81] Zachary Reynolds, Abhinandan Jayanth, Ugur Koc, Adam Porter, Rajeev Raje, and James Hill. 2017. Identifying and Documenting False Positive Patterns Generated by Static Code Analysis Tools. In *2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER IP)*. IEEE, 55–61. https://doi.org/10.1109/SER-IP.2017..20

[82] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. https://doi.org/10.1145/3188720

[83] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, Florence, Italy, 598–608. https://doi.org/10.1109/ICSE.2015.76

[84] Neil Salkind. 2020. *Encyclopedia of Research Design*. SAGE Publications, Inc. https://doi.org/10.4135/9781412961288

[85] Ben Shneiderman. 1982. Designing computer system messages. *Commun. ACM* 25, 9 (1982), 610–611. https://doi.org/10.1145/358628.358639

[86] Miltiadis Siavvas, Erol Gelenbe, Dionysios Kehagias, and Dimitrios Tzovaras. 2018. Static Analysis-Based Approaches for Secure Software Development. In *Security in Computer and Information Sciences*, Erol Gelenbe, Paolo Campegiani, Tadeusz Czachórski, Sokratis K. Katsikas, Ioannis Komnios, Luigi Romano, and Dimitrios Tzovaras (Eds.). Springer International Publishing, Cham, 142–157. https://doi.org/10.1007/978-3-319-95189-8_13

[87] Programming skills. 2020. Free Core Java Online Practice Test and Preparation for Exam. Retrieved June 2020 from https://www.pskills.org/corejava.jsp

[88] Justin Smith, Lisa Nguyen Quang Do, and Emerson Rex Murphy-Hill. 2020. Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools for Security. In *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*. USENIX Association, 221–238. https://www.usenix.org/conference/soups2020/presentation/smith

[89] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2015. Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 248–259. https://doi.org/10.1145/2786805.2786812

[90] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bei-Tseng Chu, and Heather Richter. 2019. How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool. *IEEE Transactions on Software Engineering* 45, 9 (Sept. 2019), 877–897. https://doi.org/10.1109/TSE.2018.2810116

[91] Justin Smith, Chris Theisen, and Titus Barik. 2020. A Case Study of Software Security Red Teams at Microsoft. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–10. https://doi.org/10.1109/VL/HCC50065.2020.9127203

[92] Mohammad Tahaei and Kami Vaniea. 2019. A Survey on Developer-Centred Security. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 129–138. https://doi.org/10.1109/EuroSPW.2019.00021

[93] Tyler W. Thomas, Heather Lipford, Bill Chu, Justin Smith, and Emerson Murphy-Hill. 2016. What Questions Remain? An Examination of How Developers Understand an Interactive Static Analysis Tool. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*. USENIX Association, Denver, CO, 7 pages. https://www.usenix.org/system/files/conference/soups2015/wsiw16_paper_thomas.pdf

[94] V. Javier Traver. 2010. On Compiler Error Messages: What They Say and What They Mean. *Adv. in Hum.-Comp. Int.* 2010, Article 3 (Jan. 2010), 26 pages. https://doi.org/10.1155/2010/602570

[95] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. 2014. ALETHEIA: Improving the Usability of Static Security Analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 762–774. https://doi.org/10.1145/2660267.2660339

[96] Martin Ukrop, Lydia Kraus, Vashek Matyas, and Heider Ahmad Mutleq Wahsheh. 2019. Will You Trust This TLS Certificate? Perceptions of People Working in IT. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 718–731. https://doi.org/10.1145/3359789.3359800

[97] Dirk van der Linden, Pauline Anthonysamy, Bashar Nuseibeh, Thein Than Tun, Marian Petre, Mark Levine, John Towse, and Awais Rashid. 2020. SchröDinger's Security: Opening the Box on App Developers' Security Rationale. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 149–160. https://doi.org/10.1145/3377811.3380394

[98] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering* 25, 2 (March 2020), 1419–1457. https://doi.org/10.1007/s10664-019-09750-5

[99] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (Campobasso). IEEE, 38–49. https://doi.org/10.1109/SANER.2018.8330195

[100] Daniel Votipka, Desiree Abrokwa, and Michelle L. Mazurek. 2020. Building and Validating a Scale for Secure Software Development Self-Efficacy. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–20. https://doi.org/10.1145/3313831.3376754

[101] Charles Weir, Ingolf Becker, James Noble, Lynne Blair, M. Angela Sasse, and Awais Rashid. 2020. Interventions for long-term software security: Creating a lightweight program of assurance techniques for developers. *Software: Practice and Experience* 50, 3 (2020), 275–298. https://doi.org/10.1002/spe.2774

[102] William R. Nichols and Thomas Scanlon. 2018. *DoD Developer's Guidebook for Software Assurance*. Technical Report. Software Engineering Institute - Carnegie Mellon University. 111 pages. https://resources.sei.cmu.edu/asset_files/SpecialReport/2018_003_001_538761.pdf

[103] Brett Williams, Andrys Onsman, and Ted Brown. 2010. Exploratory factor analysis: A five-step guide for novices. *Australasian Journal of Paramedicine* 8, 3 (2010), 1–13. https://doi.org/10.33151/ajp.8.3.93

[104] Michael S Wogalter. 2019. Communication-Human Information Processing (C-HIP) Model in Forensic Warning Analysis. In *Proceedings of the 20th Congress of the International Ergonomics Association (IEA 2018)*, Sebastiano Bagnara, Riccardo Tartaglia, Sara Albolino, Thomas Alexander, and Yushi Fujita (Eds.). Springer International Publishing, Cham, 761–769. https://doi.org/10.1007/978-3-319-96080-7_92

[105] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories* (Buenos Aires, Argentina) *(MSR '17)*. IEEE Press, 334–344. https://doi.org/10.1109/MSR.2017.2

[106] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman M Sadeh, Steven M Bellovin, and Joel R Reidenberg. 2017. Automated Analysis of Privacy Requirements for Mobile Apps. In *NDSS*. Internet Society, 15 pages. https://doi.org/10.14722/ndss.2017.23034

Mohammad Tahaei, Kami Vaniea, Konstantin Beznosov, and Maria Wolters

A SAT USAGE PLOTS

Plot contexts are described within the main document, in particular in Section 4.1. We include them here for more details and as a point of reference for future studies.
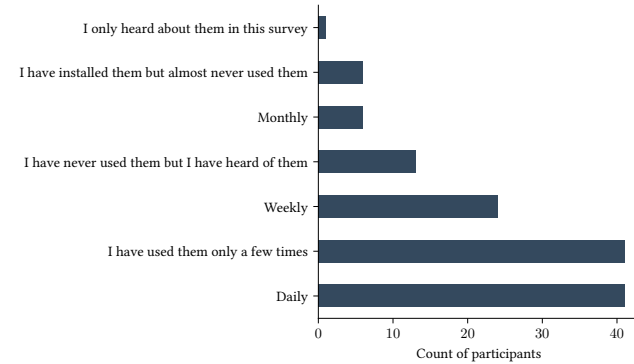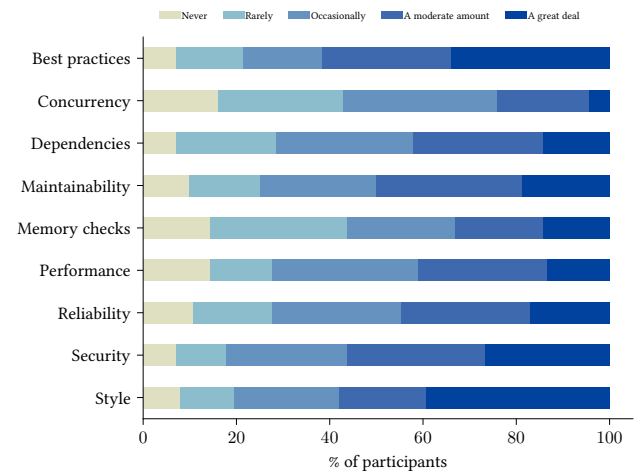
Figure 6: Answers to "how often do you use static analysis tools for any software development purpose, not just security?"
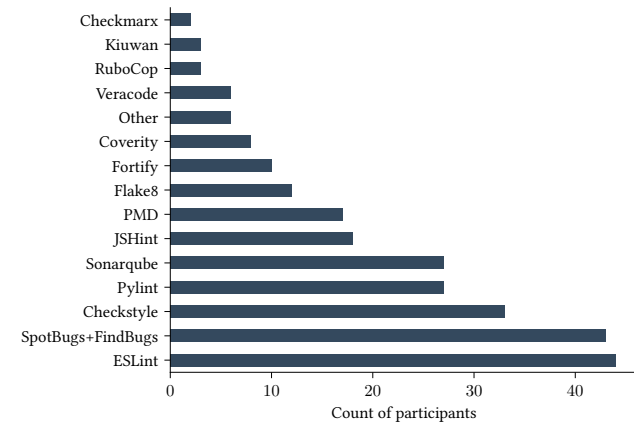
Figure 7: Answers to "which static analysis tools do you currently use?" SpotBugs and FindBugs are merged into one bar as SpotBugs is a successor of FindBugs. ESLint and JSHint are JavaScript specific, Checkstyle is primary designed for coding standards and style, Pylint and Flake8 are Python specific, and RuboCop is for Ruby programs. PMD, Fortify, Coverity, Veracode, Checkmarx have security rules for Java.
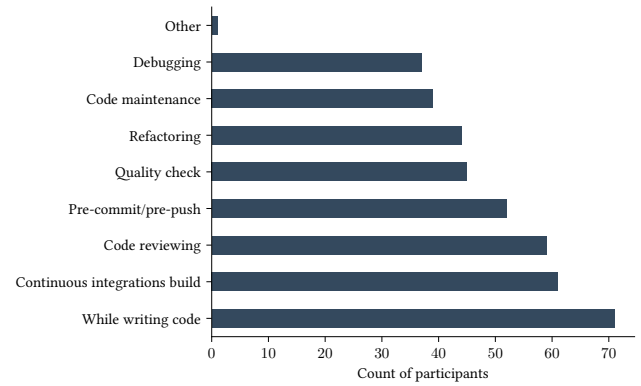
Figure 8: Answers to "in what stage of software development do you use static analysis tools?"

Figure 9: Answers to "How often do you use static analysis tools to find the following types of issues?"
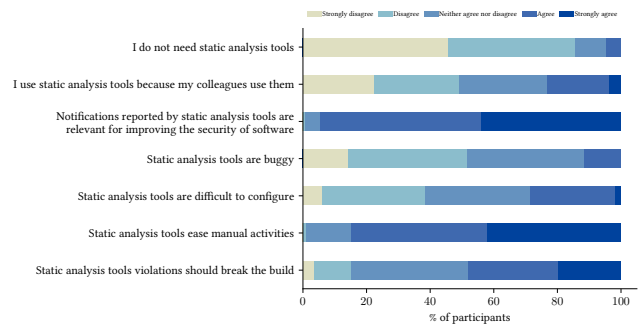
Figure 10: Answers to "please rate how much you agree or disagree with the following statements." Items are are taken from Vassallo et al [98].

B LIKERT ITEMS

Table 9 shows all Likert items that the participants saw after each sample code with the factor loading, mean, and standard deviation per condition.

Table 9: Analysis of Likert items. FL stands for factor loading from the factor analysis. Three columns with conditions represent mean (μ) and standard deviation (σ) per item across the four tasks.

		FL	Control		SonarQube		SpotBugs	
			μ	σ	μ	σ	μ	σ
Attitudes towards the notifications (Cronbach's alpha = .87)								
1	I find the notification difficult to understand (reversed).	.58	3.03	1.06	3.87	0.96	3.53	1.18
2	The notification provided enough information to understand the issues.	.80	2.48	1.10	4.01	0.90	3.98	0.93
3	I need additional information to understand the issues (reversed).	.50	2.58	1.18	3.54	1.21	3.42	1.28
4	The notification helped me understand the issues.	.75	2.92	1.08	4.06	0.73	3.88	0.95
5	The notification's descriptions are just like how I would describe the issues.	.80	2.42	1.20	3.67	0.92	3.58	1.02
6	The notification' language sounds natural to me.	.58	3.49	1.04	3.97	0.71	3.81	0.99
7	The notification tell me why the issues could be important.	.71	2.89	1.25	4.13	0.83	4.19	.0.92
8	The notification is consistent with other notifications I receive in my prior software development tasks.	.55	2.97	0.96	3.67	0.91	3.48	1.00
Prior knowledge and experience (Cronbach's alpha = .78)								
9	I have experience in writing {client server requests, SQL queries, code for databases, encryption code}.	.58	3.45	1.20	3.60	1.34	3.59	1.30
10	I used my prior knowledge about the issues to choose the code snippet solution.	.73	3.88	0.94	3.52	1.11	3.58	1.24
11	My prior knowledge about the issues was more useful than the notification content when selecting a code snippet solution.	.74	3.71	1.03	3.09	1.23	3.16	1.23
Importance and severity of the vulnerability (Cronbach's alpha = .69)								
12	The issues are of low severity (reversed).	.81	3.89	0.89	4.07	1.12	3.99	1.07
13	It is likely that an attacker can exploit the issues.	.63	3.88	0.90	4.06	1.10	4.26	0.93
14	I would try to fix the issues if this was a real-world project.	.46	4.30	0.83	4.52	0.69	4.41	0.90
Confidence in the solution (Cronbach's alpha = .78)								
15	I am confident that the code snippet solution I chose would solve the issues.	.67	3.42	1.18	3.71	1.08	3.85	1.10
16	I understand how the code snippet solution fixes the issues.	.63	3.66	1.05	3.90	0.93	3.99	1.07
Items that didn't belong to any factors								
17	The issues are rare in software projects (reversed).	<.40	3.66	0.96	3.94	0.83	3.72	0.92
18	I have encountered the notification before.	<.40	2.29	0.95	2.97	1.25	3.58	1.32