API Blindspots: Why Experienced Developers Write Vulnerable Code

Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad\textsuperscript{\alpha}, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong\textsuperscript{\nu}, Justin Cappos\textsuperscript{\nu}, Yuriy Brun\textsuperscript{\nu}, Natalie C. Ebner

University of Florida , Auto1.inc , New York University , University of Massachusetts Amherst
daniela@ece.ufl.edu, [lintian0527, rahmann, donovanellis, elianyperez, rabo, natalie.ebner]@ufl.edu, rad@akefirad.com, lad278@nyu.edu, jcappos@nyu.edu, brun@cs.umass.edu

ABSTRACT
Despite the best efforts of the security community, security vulnerabilities in software are still prevalent, with new vulnerabilities reported daily and older ones stubbornly repeating themselves. One potential source of these vulnerabilities is shortcomings in the used language and library APIs. Developers tend to trust APIs, but can misunderstand or misuse them, introducing vulnerabilities. We call the causes of such misuse blindspots. In this paper, we study API blindspots from the developers’ perspective to: (1) determine the extent to which developers can detect API blindspots in code and (2) examine the extent to which developer characteristics (i.e., perception of code correctness, familiarity with code, confidence, professional experience, cognitive function, and personality) affect this capability. We conducted a study with 109 developers from four countries solving programming puzzles that involve Java APIs known to contain blindspots. We find that (1) The presence of blindspots correlated negatively with the developers’ accuracy in answering implicit security questions and the developers’ ability to identify potential security concerns in the code. This effect was more pronounced for I/O-related APIs and for puzzles with higher cyclomatic complexity. (2) Higher cognitive functioning and more programming experience did not predict better ability to detect API blindspots. (3) Developers exhibiting greater openness as a personality trait were more likely to detect API blindspots. This study has the potential to advance API security in (1) design, implementation, and testing of new APIs; (2) addressing blindspots in legacy APIs; (3) development of novel methods for developer recruitment and training based on cognitive and personality assessments; and (4) improvement of software development processes (e.g., establishment of security and functionality teams).

1. INTRODUCTION
Despite efforts by the security community, software vulnerabilities are still prevalent in all types of computer devices [56]. Symantec Internet Security reported that 76% of all websites scanned in 2016 contained software vulnerabilities and 9% of those vulnerabilities were deemed critical [56]. According to a 2016 Vera-code report [53] on software security risk, 61% of all web applications contained vulnerabilities that fell into the Open Web Application Security Project (OWASP) Top 10 2013 vulnerability categories [39] (e.g., information leakage: 72%, flawed cryptographic implementations: 65%, carriage-return-line-feed (CRLF) injection: 53%). Further, 66% of the vulnerabilities represented programming practices that failed to avoid the “top 25 most dangerous programming errors” identified by CWE/SANS [12]. In addition, new instances of existing, well-known vulnerabilities, such as SQL injections and buffer overflows, are still frequently reported in vulnerability databases [50, 26]. These data affirm that current software security awareness efforts have not eradicated these problems in practice.

A contributing factor in the introduction of software vulnerabilities may be the way developers view the programming language resources they routinely use. APIs provide developers with high-level abstractions of complex functionalities and are crucial in scaling software development. Yet, studies on API usability [46, 47] and code comprehension [25] show that developers experience a number of challenges while using APIs, such as mapping developer-specific requirements to proper usage protocols, making sense of internal implementation and related side effects, and deciding between expert opinions. Further, misunderstandings in developers’ use of APIs are frequently the cause of security vulnerabilities [9, 14, 45]. Developers often blindly trust APIs and their misunderstanding of the way API functions are called may lead to blindspots, or oversights regarding a particular function usage (e.g., assumptions, results, limitations, exceptions).

More significantly, when developers use an API function, they may behave as if they are outsourcing any security implications of its use [37]. That is, they do not see themselves as responsible for the correct usage of the function and any possible resulting security consequences.

An API security blindspot is a misconception, misunderstanding, or oversight [9] on the part of the developer when using an API function, which leads to a violation of the recommended API usage protocol with possible introduction of security vulnerabilities. Blindspots can be caused by API functions whose invocations have security implications that are not readily apparent to the developer. It is analogous to the concept of a car blindspot, an area on the side of a car that is not visible to the driver that can lead to accidents. For an example of an API blindspot, consider the strcpy() function from the C standard library. For almost three decades [41], this function has been known to lead to a buffer overflow vulnerability if developers do not check and match sizes of the destination and source arrays. Yet, developers tend to have a blindspot with respect
to this function. In a recent study [37], a developer who could not detect a buffer overflow in a programming scenario mentioned that “It’s not straightforward that misusing strcpy() can lead to very serious problems. Since it’s part of the standard library, developers will assume it’s OK to use. It’s not called unsafe strcpy() or anything, so it’s not immediately clear that that problem is there.”

In this paper, we present an empirical study of API blindspots from the developers’ perspective, and consider personal characteristics that may contribute to the development of these blindspots. Our study goals were to: (1) determine developers’ ability to detect API blindspots in code and (2) examine the extent to which developer characteristics (i.e., perception of code correctness, familiarity with code, confidence in correctly solving the code, professional experience, cognitive function, personality) affected this capability. We also explored the extent to which API function or programming scenario characteristics (i.e., category of API function and cyclomatic complexity of the scenario) contributed to developers’ ability to detect blindspots.

We recruited 109 developers, including professional developers and senior undergraduate and graduate students (professionals = 70, students = 39, mean age = 26.4, 80.7% male). Developers worked online on six programming scenarios (called puzzles) in Java. Each puzzle contained a short code snippet simulating a real-world programming scenario. Four of the six puzzles contained one API function known to cause developers to experience blindspots. The other two puzzles involved an innocuous API function. Puzzles were developed by our team and were based on API functions commonly reported in vulnerability databases [36, 49] or frequently discussed in developer forums [51]. The API functions considered addressed file and stream handling, cryptography, logging, SQL operations, directory access, regular expressions (regex), and process manipulation. Following completion of each puzzle, developers responded to one open-ended question about the functionality of the code and one multiple-choice question that captured developers’ experience with programming did not predict their ability to detect API blindspots.

These results have the potential to inform the design of APIs that are inherently more secure. For example, testing and validation of API functions should take into account potential security blindspots developers may have, particularly for certain types of API functions (e.g., I/O). Furthermore, since our data suggest that experience and cognition may not predict developers’ ability to detect API blindspots, it corroborates the validity of the emerging practice of establishing separate functionality and security development teams. Separate teams for these domains may be a better strategy to assure secure software development than sole reliance on one group of experts to simultaneously address both aspects.

The remainder of this paper is organized as follows. Section 2 reports on the study methodology and the development of the puzzles. Section 3 assesses the results, while Section 4 discusses some of the implications of these findings. Section 5 places this study in the context of related work, and Section 6 summarizes its primary contributions.

2. METHODOLOGY

This section presents the study methodology, describing recruitment, participant management, and procedures. Data collection took place between December 2016 and November 2017.

2.1 Participants

This study, approved by the University of Florida IRB, targeted developers who actively worked with Java. These individuals were recruited from the United States, Brazil, India, Bangladesh, and Malaysia via a number of recruitment mechanisms, including flyers and handouts disseminated throughout the university campus, particularly in locations frequented by students and professionals with programming experience (e.g., Computer Science and Engineering departments), social media advertisements (i.e., Facebook, Twitter, and LinkedIn), ads on online computer programming forums, Computer Science/Engineering department groups, and contacts via the authors’ personal networks of computer programmers at universities and software development companies in the United States, Brazil, India, Bangladesh, and Malaysia. We also used a word-of-mouth recruiting technique, which gave participants the option to refer friends or colleagues. Participants were informed that the purpose of the study was to investigate how developers interpret and reason about code. As we aimed to have developers work on the programming tasks as naturally as possible, without any priming or nudging towards software security aspects, we did not explicitly mention that code security was the metric of interest. Figure 1 summarizes the demographic information of participating developers.

As shown in Figure 1, developers in the final sample size (N = 109) ranged between the ages of 21 and 52 years (M = 26.67, SD = 5.28) and were largely male (n = 88, 80.7%). The sample was composed of 70 (64.2%) professional developers and 39 (35.8%) senior undergraduate or graduate students in Computer Science and Computer Engineering though in this paper, we collectively refer to all participants as “developers”. The large majority of developers (n = 83, 82.5%) had been programming in Java for two or more years, and almost all developers reported at least a working knowledge of Java (n = 101, 97.1%). Student participants self-reported a relatively high programming experience (M = 5.8 years, SD = 5.8), probably because they had been programming before entering university or had been students for more than six years (e.g., PhD students).

We received a total of 168 emails from interested developers, 33 (19.6%) of which were not included in the study because they never
sioned the informed consent form or signed the form but did not continue with the assessment. The remaining 135 developers received a personalized link to the study assessment, which was hosted online on the Qualtrics platform. We had to discard data from 26 (19.3%) developers because of incomplete entries or technical/browser incompatibility issues related to the audio recording (see details below). Unless otherwise stated, we report our results based on a sample of 109 developers, who proceeded through all study procedures as instructed and completed the tasks with valid responses.

2.2 Procedure

After initial contact with interested developers, an online screening questionnaire determined study eligibility (e.g., sufficient knowledge with Java, fluency in the English language, age over 18 years). Eligible developers received a digital informed consent form, which disclosed study procedures, the minimal risk from participating, and potential data privacy and anonymity issues. After providing their digital signature, developers received a personalized link to the online instrument. Each developer was assigned a unique identifier to assure confidentiality. Developers were strongly encouraged to complete the study in two separate sittings to counteract possible fatigue effects (one sitting to work on the puzzles and complete the demographic questionnaire, and the other sitting to complete the psychological/cognitive assessment). Student developers were compensated with a US$20 Amazon gift card, while professionals received a US$50 Amazon gift card, as professional developers had a larger financial incentive in consideration of their relatively high-paying jobs and their more limited availability, as approved by our IRB. The study procedure comprised five parts. The first part (Puzzles) involved responding to the programming puzzles and related questions (see Section 2.3). The second part (Demographics) asked basic demographic questions about the subject, including age, gender, race/ethnicity, education, field of study, employment status, and primary language.

The third part (Professional Experience and Expertise) included questions about the developers’ technical proficiency and years of programming experience in six commonly used programming languages (i.e., Java, Python, C/C++, PHP, Visual Basic.Net, and JavaScript). A free-text response field was provided for developers to record their preferred programming language, if it was not listed. Developers also indicated their level of knowledge in and experience with 17 programming concepts and technologies identified from the literature and via job postings for software developers [6, 30] (e.g., SQL/MySQL, Cryptography, File compression, Networking, HTTP/HTTPS, I/O operations).

The fourth part (Personality Assessment) used the Big Five Inventory (BFI) questionnaire to measure aspects of personality [29]. This questionnaire contains 44 items to assess five personality dimensions: Openness, Conscientiousness, Extraversion, Agreeableness, and Neuroticism. Developers rated the extent to which they endorsed each personality statement on a Likert scale (1 = disagree strongly; 5 = agree strongly). We computed the sum score across all items for each of the five personality dimensions.

The fifth part (Cognitive Assessment) comprised two instruments: the Oral Symbol Digit Test from the NIH Toolbox [21] and the Brief Test of Adult Cognition by Telephone (BTACT) [58]. Figure 2 illustrates the Oral Symbol Digit Test. This test is a brief measure of processing speed and working memory. In this task, developers were presented with a coding key containing nine abstract symbols, each paired with a number between 1 and 9. They were then given 120 seconds to call out as many numbers that went with the corresponding symbols, in the order presented and without skipping any. The BTACT is a battery of cognitive processing tasks for adults of different ages and takes approximately 20 minutes to complete. The BTACT sub-tests refer to episodic verbal memory, working memory, verbal fluency, inductive reasoning, and processing speed. Figure 3 presents instructions for the BTACT Word List Recall task, which measures immediate and delayed episodic memory for verbal material. This particular task asked developers to recall a number of spoken words. The Oral Symbol Digit Test and the BTACT were chosen based on the cognitive processes (e.g., reasoning, working memory, processing speed) developers likely use when working on code. Traditionally, the Oral Symbol Digit Test and the BTACT are administered in-person and over the phone, respectively. Given the online format of our study, we implemented browser-based audio recordings of the two measures. In particular, audio narrations for all the tasks instructions were created, with calculated timings, vocal inflections, and pauses. Formal time limits were maintained. To capture oral responses, we built an audio recording plugin leveraging the Qualtrics JavaScript API. All task modifications underwent pilot testing to ensure that content and response sensitivity was maintained. As part of the study infrastructure, recorded audio files were sent to a secure and encrypted study server and were stored in an anonymized fashion. Trained coders coded these files for performance in the various tasks. For
This process began with a literature review to
Puzzle creation.

study, as we aimed to recruit from a diverse pool of developers.

features made Java a good choice of programming language for our
visit Stack Overflow with programming related Q&As [52]. These
the third most popular technology by developers who frequently
language in GitHub repositories after Javascript [22] and was voted
oriented programming. It is the second most used programming
resources has made it a popular choice for those learning object-
erprise [15]. Besides being popular among professional develop-
ers, Java's wide availability of toolkits, tutorials, and online/offline
ability databases [36, 49], HPE's Software Security Taxonomy [19,
and the Java API official documentation [28]. We also lever-
egaged programming Q&A forums, such as Stack Overflow [51] to
select commonly discussed API functions. We did not look for
candidate blindspot functions in bug repositories because we did
not want developers in our study to fix bugs in code. Instead, our
aim was to analyze whether developers would detect improper API
usage to infer the insecure behaviors to which it may lead. Thus,
all the code snippets were free from bugs and compilation errors,
and were compatible with Java standard edition version 7 or higher.
Our API function selection process included functions from differ-
cent categories, including I/O, cryptography, SQL, and string.

We initially identified 61 API function candidates and created 61
 corresponding puzzles, each targeting one particular function. This
pool encompassed a variety of Java API misuse scenarios, includ-
ing file I/O operations, garbage collection, de/serialization, cryp-
tography, secure connection establishment, command line argu-
ments/user inputs processing for database query, logging, user au-
thentication, and multithreading.

Each puzzle contained four parts: (1) the puzzle scenario itself;
(2) an accompanying code snippet; (3) a question about the puz-
le's functionality, and (4) a multiple-choice question, which, for
blindspot puzzles was implicitly related to code security and for
non-blindspot puzzles related to code functionality. Developers' 
accuracy on the multiple-choice question served as the central out-
come measure. It captured the developers' understanding of the
blindspot in the code.

Puzzle review and final selection. Three co-authors, who had not
created the puzzles, independently reviewed the initial set of 61
blindspot puzzles, together with eight non-blindspot puzzles, to en-
sure puzzle accuracy, legibility, coherence, and relevance to real-
life programming situations. The specific criteria used for puzzle
approval were:

1. Is the scenario clear and realistic?
2. Is the code snippet clear and concise (maximum one screen)?
3. Does the code snippet compile and run if provided with the
necessary Java packages?
4. Does the choice of API function contribute to diversity in the
puzzle set (API function category, blindspot vs. non-blindspot
function, blindspot by function omission vs. presence, and number of parameters)?
5. Does the multiple-choice question have only one answer with-
out ambiguity?
6. For blindspot puzzles, does the multiple-choice question address the security implications subtly without priming developers about security concerns?

7. For blindspot puzzles, is there a way to rewrite the puzzle to address the security vulnerability, thus avoiding the blindspot?

To be contained in the final pool, puzzles had to be independently approved by all three reviewers.

The final set comprised 16 blindspot puzzles and eight non-blindspot puzzles, which varied in the following categories: (1) blindspot vs. non-blindspot; (2) API usage category; and (3) cyclomatic complexity.

**Blindspot vs. non-blindspot.** We included non-blindspot puzzles as a control and to cover the security focus of the study. Blindspot puzzles were bug-free and functionally correct, but could cause a blindspot in developers when they used them, thus having the potential to cause developers to introduce one of the following vulnerabilities in code: (1) arbitrary code/command injection; (2) DoS (exhaustion of local resources); (3) time-of-check-to-time-of-use (TOCTTOU); (4) sensitive data disclosure; (5) broken or flawed cryptographic implementation; and (6) insecure file and I/O operations.

**API usage category.** The puzzles referred to three different API usage contexts: (1) I/O, involving operations, such as reading and writing from/to streams and files, internal memory buffers, and networking activity; (2) Crypto, involving functions handling cryptographic operations, such as encryption, decryption, and key agreement; and (3) String, involving functions that perform string processing or manipulation, or queries and user input.

**Cyclomatic complexity [31].** Puzzles varied in their cyclomatic complexity, defined as a quantitative measure of the number of linearly independent paths in the source code. We classified the cyclomatic complexity of each puzzle into one of three levels: an integer value of low (cyclomatic complexity of 1–2), medium (cyclomatic complexity of 3–4), or high (cyclomatic complexity > 4) complexity.

We divided the final set of 24 puzzles into four subsets, each set containing six puzzles, four with a blindspot and two without a blindspot. This counterbalancing scheme ensured that each puzzle set was comparable regarding representation of API category and cyclomatic complexity. Statistical analysis found no effects for puzzle sets as covariate, confirming successful counterbalancing. We assigned each developer randomly to one of the four puzzle sets.

Figure 4 illustrates a blindspot puzzle, involving a Java Runtime API usage. The puzzle scenario was presented to developers as follows:

> "You are asked to review a utility method written for a web application. The method, `setDate(String)`, changes the date of the server. It takes a `String` as the new date ("dd-mm-yyyy" format), attempts to change the date of the server, and returns `true` if it succeeded, and `false` otherwise. Consider the snippet of code below (assuming the code runs on a Windows operating system) and answer the following questions, assuming that the code has all required permissions to execute."

After presenting the code snippet, developers were asked which of the following statements would be correct if the `setDate()` method was invoked with an arbitrary `String` value as the new date:

- a. If the given `String` value does not conform to the “dd-mm-yyyy” format, an exception is thrown.
- b. The `setDate()` method cannot change the date.
- c. The `setDate()` method might do more than change the date.
- d. The return value of the `waitFor()` method is not interpreted correctly (lines 14–17).
- e. The web application will crash.

The correct answer is option ‘c’. A close inspection of the code shows that the `Runtime.getRuntime().exec()` method executes, in a separate process, the specified string command (line 10) which is provided by the `setDate()` method. The `setDate()` method takes a `String` type argument and does not implement any input sanitization and validation, which makes it vulnerable to format string injection attacks. For example, calling the `setDate()` method with “10-12-2015 & & shutdown /s” as the argument changes the date and turns off the server. Either the argument for `setDate()` method has to be sanitized or its type should be an instance of the `Date` class, which can be formatted as a `String` type before passing to the `Runtime.getRuntime().exec()` method. As the outcome of the program (executing in a benign or malicious fashion) depends solely on the (un)sanitized input of the `Runtime.exec()` method, the blindspot API function for this puzzle is `Runtime.exec()`.

Table 1 details the complete list of puzzles used in the study with information about the puzzle’s vulnerability, the API usage context, and the Java API function targeted for both blindspot and non-blindspot puzzles.

After completion of a puzzle and related security questions, developers responded to the following four questions about their puzzle perceptions using a Likert scale (1 = not at all to 10 = very): (1) Difficulty (How difficult was this scenario?); (2) Clarity (How clear was this scenario?); (3) Familiarity with the API functions presented in the code snippet (How familiar were you with the functions in this scenario?); and (4) Confidence (How confident were you that you solved the scenario correctly?).
Table 1: Overview of the final puzzle set with information about puzzle vulnerability, API usage context, and Java API function targeted in each puzzle.

<table>
<thead>
<tr>
<th>Has Blindspot</th>
<th>Vulnerability (if any)</th>
<th>Description</th>
<th>API Usage Context</th>
<th>Targeted (non) Blindspot API function</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>TOCTTOU/race condition</td>
<td>A program that performs two or more file operations on a single file name or path name creates a race window between the two file operations. Thus, File.createTempFile() may overwrite an existing file even after the overwrite flag is set to false. File.renameTo() relies solely on file names for identification, which does not guarantee that the file renamed is the same file that was opened, processed, and closed, thus being vulnerable to the TOCTTOU vulnerability. JVM does not guarantee the timing for garbage collection of an object. Malicious subclasses that override the Object.finalize() method can resurrect objects meant for garbage collection.</td>
<td>I/O</td>
<td>java.io.File.createNewFile()</td>
</tr>
<tr>
<td>YES</td>
<td>TOCTTOU/race condition</td>
<td></td>
<td>I/O</td>
<td>java.io.File.renameTo()</td>
</tr>
<tr>
<td>YES</td>
<td>Resurrectable object</td>
<td></td>
<td>I/O</td>
<td>java.lang.Object.finalize()</td>
</tr>
<tr>
<td>YES</td>
<td>Ambiguous return value</td>
<td></td>
<td>I/O</td>
<td>java.util.zip.Zipentry.setSize()</td>
</tr>
<tr>
<td>YES</td>
<td>Flawed cryptographic implementation</td>
<td></td>
<td>Crypto</td>
<td>javax.crypto.Cipher.doFinal()</td>
</tr>
<tr>
<td>YES</td>
<td>Flawed cryptographic implementation</td>
<td></td>
<td>Crypto</td>
<td>javax.crypto.Cipher.update()</td>
</tr>
<tr>
<td>YES</td>
<td>Flawed cryptographic implementation</td>
<td></td>
<td>Crypto</td>
<td>javax.crypto.Cipher.doFinal()</td>
</tr>
<tr>
<td>YES</td>
<td>Flawed cryptographic implementation</td>
<td></td>
<td>Crypto</td>
<td>javax.crypto.Cipher.update()</td>
</tr>
<tr>
<td>YES</td>
<td>Improper input validation</td>
<td></td>
<td>String</td>
<td>java.lang.Runtime.exec()</td>
</tr>
<tr>
<td>YES</td>
<td>Improper input validation</td>
<td></td>
<td>String</td>
<td>new java.lang.ProcessBuilder()</td>
</tr>
<tr>
<td>YES</td>
<td>Improper input validation</td>
<td></td>
<td>String</td>
<td>java.sql.PreparedStatement.setString()</td>
</tr>
<tr>
<td>YES</td>
<td>Improper input validation</td>
<td></td>
<td>String</td>
<td>javax.naming.directory.DirContext.search()</td>
</tr>
<tr>
<td>YES</td>
<td>Improper input validation</td>
<td></td>
<td>String</td>
<td>java.util.regex.Matcher.matches()</td>
</tr>
<tr>
<td>YES</td>
<td>Improper input validation</td>
<td></td>
<td>String</td>
<td>java.util.logging.Logger.info()</td>
</tr>
<tr>
<td>YES</td>
<td>Disclosure of sensitive information</td>
<td></td>
<td>I/O</td>
<td>java.io.File.deleteOnExit()</td>
</tr>
<tr>
<td>YES</td>
<td>Disclosure of sensitive information</td>
<td></td>
<td>I/O</td>
<td>java.nio.file.Files.write()</td>
</tr>
<tr>
<td>NO</td>
<td>N/A</td>
<td></td>
<td>I/O</td>
<td>java.io.File.createNewFile()</td>
</tr>
<tr>
<td>NO</td>
<td>N/A</td>
<td></td>
<td>I/O</td>
<td>java.io.File.renameTo()</td>
</tr>
<tr>
<td>NO</td>
<td>N/A</td>
<td></td>
<td>I/O</td>
<td>java.io.InputStream.read()</td>
</tr>
<tr>
<td>NO</td>
<td>N/A</td>
<td></td>
<td>I/O</td>
<td>java.util.zip.Zipentry.setSize()</td>
</tr>
<tr>
<td>NO</td>
<td>N/A</td>
<td></td>
<td>I/O</td>
<td>java.io.File.listFiles()</td>
</tr>
<tr>
<td>NO</td>
<td>N/A</td>
<td></td>
<td>Crypto</td>
<td>javax.crypto.Cipher.doFinal()</td>
</tr>
<tr>
<td>NO</td>
<td>N/A</td>
<td></td>
<td>Crypto</td>
<td>javax.crypto.Cipher.update()</td>
</tr>
<tr>
<td>NO</td>
<td>N/A</td>
<td></td>
<td>Crypto</td>
<td>java.nio.file.Files.close()</td>
</tr>
</tbody>
</table>

320  Fourteenth Symposium on Usable Privacy and Security  USENIX Association
In sum, we collected the following measures from the developers: (1) responses to puzzles; (2) developer-reported perceptions of puzzle difficulty, clarity, familiarity with puzzle functions, and confidence in solving the puzzle; (3) demographic information; (4) programming experience and skills; (5) personality traits; and (6) cognitive functioning scores.

**Debriefing.** All developers were debriefed at the end of the study about its true purpose and presented with the correct solutions for each puzzle they had worked on, including the rationale for the correct answer. The study ended by soliciting feedback about the study and processing compensation.

### 3. DATA ANALYSIS AND RESULTS

This section presents the results of the study and the findings that emerged from the data. We used the statistical software package STATA 14.0 for data analysis. As described in Section 1, the study goals were to (1) determine developers’ ability to detect API blindspots in code and (2) examine the extent to which developer characteristics affected this capability. In particular, we tested the following hypotheses:

**H1:** Developers are less likely to correctly solve puzzles with API functions containing blindspots than puzzles with innocuous functions (non-blindspot puzzles).

**H2:**
- **a:** Developers perceive puzzles with API functions containing blindspots as more difficult than non-blindspot puzzles.
- **b:** Developers perceive puzzles with API functions containing blindspots as less clear than non-blindspot puzzles.
- **c:** Developers perceive puzzles with API functions containing blindspots as less familiar than non-blindspot puzzles.
- **d:** Developers are less confident about their puzzle solution when working on puzzles with API functions containing blindspots than non-blindspot puzzles.

**H3:** Higher cognitive functioning (reasoning, working memory, processing speed) in developers is associated with greater accuracy in solving puzzles with API functions containing blindspots.

**H4:** Higher levels of professional experience and expertise in developers are associated with greater accuracy in solving puzzles with API functions containing blindspots.

**H5:** Higher levels of conscientiousness and openness, and lower levels of neuroticism and agreeableness in developers are associated with greater accuracy in solving puzzles with API functions containing blindspots.

We used multilevel modeling to test H1 and H2a–d and ordinal logistic regression to test H3, H4, and H5 (see details below).

The main purpose of our analyses for all hypotheses was to determine the significance of specific effects (e.g., effect of a given personality trait on accuracy for blindspot puzzles), rather than identifying the best model to represent our data. Therefore, we did not apply a model comparison approach in our central analyses. In the exploratory analyses in Section 3.1, however, we were interested in determining the extent to which adding moderators (i.e., API usage type, cyclomatic complexity) enhanced the fit of our model, compared to the model originally tested under H1. In these instances, we report relevant goodness of fit indices (Akaike Information Criterion [AIC] and Bayesian Information Criteria [BIC]) [8].

Unless mentioned otherwise, we considered effects with p-values smaller than 0.05 as significant.

#### 3.1 H1: Puzzle accuracy for blindspot vs. non-blindspot puzzles

We used multilevel logistic regression to test H1, accommodating for (1) the hierarchical data structure in which each set of six puzzles (level-1) was nested within each developer (level-2) and (2) the dichotomous outcome variable puzzle accuracy (1 = correct answer, 0 = incorrect answer). The independent variable was the presence of a blindspot (0 = no blindspot; 1 = blindspot). In this model, we also considered the random effect of the intercept to accommodate for inter-individual differences in overall puzzle accuracy. Presence of a blindspot had a significant effect on puzzle accuracy (Wald $\chi^2(2) = 20.60, p < .001$, Table 2), supporting H1 that developers were less likely to correctly solve puzzles with API functions containing blindspots than in those puzzles without blindspots.

In an exploratory fashion, we examined the extent to which (1) API usage type (i.e., I/O, Crypto, and String, see Section 2.3) and (2) puzzle cyclomatic complexity qualified the observed effect of the presence of blindspot on puzzle accuracy. The small number of puzzles in each set limited our capability to examine those two predictors in a single model. Therefore, we ran these exploratory analyses in two separate models, one for API usage type and the other for puzzle cyclomatic complexity. We used Wald tests to determine the significance of the main effects and interactions. To control for family-wise type-I error inflation due to multiple dependent models (i.e., models that share the same dependent variable), we applied Bonferroni correction for the threshold of the p-value to determine statistical significance in these exploratory analyses ($p < 0.025$).

**API usage type.** We added the categorical variable API usage type (1 = I/O, 2 = Crypto, 3 = String) and its interaction with the presence of blindspot as predictors in the model. Both the AIC and BIC were smaller for this model with the added moderator than for the H1 model (Table 2), suggesting a better goodness of fit when adding API usage as a moderator into the model. The main effect of presence of blindspot was not significant (Wald $\chi^2(1) = 0.91, p = 0.34$), but the main effect of API usage type
Bonferroni correction of the types of API usage: I/O, Crypto, and String. The y-axis shows the presence of blindspot on puzzle accuracy. The x-axis shows the three interaction effect of API usage type and presence of blindspot puzzles with API functions that involved the other two usage types (i.e., Crypto, String).

Cyclomatic complexity. We added the categorical variable cyclomatic complexity (1 = low, 2 = medium, 3 = high) and its interaction with the presence of blindspot puzzles than for blindspot ones with an API function that involved I/O. Accuracy was comparable in both non-blindspot and blindspot puzzles with API functions that involved the other two usage types (i.e., Crypto, String).

3.2 H2: Developers’ perceptions for blindspot vs. non-blindspot puzzles

For H2a–d, we again used multilevel modeling to accommodate for the hierarchical data structure. The dependent variables for H2a–d were the four continuous rating dimensions (i.e., difficulty, clarity, familiarity, confidence), respectively, which we submitted to four separate models for H2a–d. For H2a, we tested for the main effect of cyclomatic complexity (1 = low, 2 = medium, 3 = high) and its interaction with the presence of blindspot (Wald χ²(2) = 24.81, p < 0.001) was significant. As shown in Figure 6, accuracy was higher for non-blindspot puzzles than for blindspot ones with an API function that involved I/O. Accuracy was comparable in both non-blindspot and blindspot puzzles with API functions that involved the other two usage types (i.e., Crypto, String).

Cognitive Function. Our analyses pertaining to H3 resulted in no significant effects for any of the three cognitive measures on blindspot puzzle accuracy (all ps > 0.10, Table 4). Thus, the data did not support H3.

Technical Experience/Expertise. As shown in Table 5, none of the three predictors of experience/expertise predicted blindspot puzzle accuracy (all ps > 0.10). Thus, the data did not support H4.

Personality Traits. As shown in Table 5, the effect of openness on blindspot puzzle accuracy was significant (p < 0.001). That is, greater openness as a personality trait in developers was associated with greater accuracy in solving blindspot puzzles. None of the other personality dimensions showed significant effects (all p > 0.09).

4. DISCUSSION

This section summarizes the study findings, discusses study strengths and limitations, and offers actionable recommendations.

4.1 Summary of findings

The goal of this study was to examine API blindspots from the developers’ perspective: (1) determine the extent to which developers can detect API blindspots in code with the goal to improve understanding of the implication blindspots have on software security; and (2) determine the extent to which developer characteristics (i.e., difficulties with code, perceptions of code clarity, familiarity with code, confidence in solving puzzles, developers’ level of cognitive functioning, their professional experience and expertise, and their personality traits) influenced developers’ ability to detect blindspots. We also explored the extent to which API usage category and cyclomatic complexity of the puzzles impacted developers’ ability to detect blindspots.
Table 2: Effect of presence of blindspot on puzzle accuracy (H1) and results of exploratory analyses on the moderation of API usage type and cyclomatic complexity on puzzle accuracy.

<table>
<thead>
<tr>
<th>Fixed Effect</th>
<th>Hypothesis 1</th>
<th>Expl. Anal. – API Usage Type</th>
<th>Expl. Anal. – Cyclomatic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presence of blindspot</td>
<td>Blindspot</td>
<td>0.44 (0.08) [0.31, 0.63]</td>
<td>0.16 (0.05) [0.09, 0.31]</td>
</tr>
<tr>
<td>API usage type</td>
<td>Crypto</td>
<td>0.33 (0.13) [0.15, 0.71]</td>
<td>0.11 (0.07) [0.04, 0.37]</td>
</tr>
<tr>
<td>Presence of blindspot × API usage type</td>
<td>Blindspot × Crypto</td>
<td>9.10 (4.50) [3.45, 23.98]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Blindspot × String</td>
<td>11.35 (7.85) [2.92, 44.04]</td>
<td></td>
</tr>
<tr>
<td>Cyclomatic complexity</td>
<td>Medium</td>
<td>1.52 (0.68) [0.64, 3.63]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>6.88 (2.62) [3.26, 14.53]</td>
<td></td>
</tr>
<tr>
<td>Presence of blindspot × Cyclomatic complexity</td>
<td>Blindspot × Medium</td>
<td>0.29 (0.15) [0.10, 0.82]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Blindspot × High</td>
<td>0.02 (0.01) [0.005, 0.08]</td>
<td></td>
</tr>
</tbody>
</table>

Random Effect

<table>
<thead>
<tr>
<th>Intercept</th>
<th>σ² (SE) 95% CI</th>
<th>σ² (SE) 95% CI</th>
<th>σ² (SE) 95% CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.43 (0.20) [0.17, 0.99]</td>
<td>0.72 (0.28) [0.34, 1.52]</td>
<td>0.54 (0.25) [0.22, 1.33]</td>
<td></td>
</tr>
</tbody>
</table>

Goodness of Fit

<table>
<thead>
<tr>
<th>AIC</th>
<th>BIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>824.13</td>
<td>837.58</td>
</tr>
<tr>
<td>794.63</td>
<td>826.01</td>
</tr>
<tr>
<td>773.26</td>
<td>804.64</td>
</tr>
</tbody>
</table>

Note. O. R. = odds ratio; SE = standard error; CI = confidence interval. We used robust standard errors to accommodate for the hierarchical data structure. The reference category is non-blindspot for “presence of blindspot”, I/O for “API usage type”, and low for “cyclomatic complexity”. Bonferroni correction was applied to p-values in the simple effect analyses for the main effect of API usage type and cyclomatic complexity and the follow-up analyses to counter inflation of type-I errors due to multiple comparison. Bold indicates significant effects at p < .05.

Table 3: Effect of presence of blindspot on developers’ perception of puzzles.

<table>
<thead>
<tr>
<th>Fixed Effect</th>
<th>H2a: Difficulty</th>
<th>H2b: Clarity</th>
<th>H2c: Familiarity</th>
<th>H2d: Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presence of Blindspot</td>
<td>B (SE) 95% CI</td>
<td>B (SE) 95% CI</td>
<td>B (SE) 95% CI</td>
<td>B (SE) 95% CI</td>
</tr>
<tr>
<td>Blindspot</td>
<td>0.16 (0.14) [-0.12, 0.43]</td>
<td>-0.01 (0.12) [-0.25, 0.23]</td>
<td>-0.10 (0.15) [-0.40, 0.19]</td>
<td>-0.11 (0.13) [-0.36, 0.15]</td>
</tr>
</tbody>
</table>

Random Effect

| Intercept | 2.27 (0.33) [1.31, 3.01] | 2.22 (0.37) [1.61, 3.07] | 1.67 (0.32) [1.15, 2.43] | 1.72 (0.37) [1.13, 2.60] |

Note. B = unstandardized regression coefficient; SE = standard error; CI = confidence interval. The reference category is non-blindspot for “presence of blindspot”. Bold indicates significant effects at p < .05.

Table 4: Effect of developers’ level of cognitive function on puzzle accuracy.

<table>
<thead>
<tr>
<th>Cognitive Function</th>
<th>Blindsport Puzzles O.R. (SE) 95% CI</th>
<th>Non-Blindspot Puzzles O.R. (SE) 95% CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reasoning</td>
<td>1.16 (0.17) [0.87, 1.54]</td>
<td>1.31 (0.21) [0.96, 1.80]</td>
</tr>
<tr>
<td>Working Memory</td>
<td>1.12 (0.08) [0.97, 1.28]</td>
<td>1.09 (0.11) [0.90, 1.33]</td>
</tr>
<tr>
<td>Processing Speed</td>
<td>1.00 (0.01) [0.99, 1.02]</td>
<td>1.01 (0.01) [0.99, 1.03]</td>
</tr>
</tbody>
</table>

Note. O.R. = odds ratio; SE = standard error; CI = confidence interval. 91 developers were included in the analysis for reasoning, 90 for working memory, and 89 developers for processing speed.

Our results confirmed H1 that developers are less likely to correctly solve puzzles with blindspots compared to puzzles without blindspots. This finding suggests that developers experience security blindspots while using certain API functions. Oliveira et al. [37] interviewed professional developers and found that they generally trust APIs. Given this general trust, even security-minded developers may not explicitly look for vulnerabilities in API functions, with the result that blindspots cause security vulnerabilities.

Our exploratory analyses suggested that the presence of blindspot particularly impacts accuracy in solving puzzles with I/O-related API functions, and with more complex programming scenarios (i.e., high cyclomatic complexity).

Our data did not support H2a-H2d, that posited developers’ perceptions of puzzle difficulty, clarity, familiarity, and confidence are associated with their ability to detect blindspots. Our results also did not support H3 that developers’ level of cognitive functioning could predict their ability to detect blindspots.

We also found no support for H4 that professional and technical experience were associated with developers’ ability to detect blindspots. This finding is in line with research on code review that showed a developer’s amount of experience does not correlate with greater accuracy or effectiveness in detecting security issues in code [16].

Our results partially support H5 as more openness as a personality trait in developers does appear to be associated with a higher likelihood to detect blindspots. Openness relates to intellectual curiosity and the ability to use one’s imagination [29]. It is plausible...
Table 5: Effect of developers’ professional expertise and personality traits on puzzle accuracy.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Blindsight Puzzles</th>
<th>Non-Blindsight Puzzles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O.R. (SE)</td>
<td>95% CI</td>
</tr>
<tr>
<td>Professional Expertise</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Years of programming</td>
<td>0.81 (0.70)</td>
<td>[0.15, 4.45]</td>
</tr>
<tr>
<td>Technical expertise</td>
<td>0.93 (0.12)</td>
<td>[0.72, 1.19]</td>
</tr>
<tr>
<td>Java skills</td>
<td>1.11 (0.15)</td>
<td>[0.85, 1.45]</td>
</tr>
<tr>
<td>Personality Traits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Agreeableness</td>
<td>0.95 (0.05)</td>
<td>[0.85, 1.05]</td>
</tr>
<tr>
<td>Conscientiousness</td>
<td>0.97 (0.05)</td>
<td>[0.88, 1.07]</td>
</tr>
<tr>
<td>Extraversion</td>
<td>0.94 (0.04)</td>
<td>[0.87, 1.01]</td>
</tr>
<tr>
<td>Neuroticism</td>
<td>0.93 (0.04)</td>
<td>[0.86, 1.01]</td>
</tr>
<tr>
<td>Openness</td>
<td>1.18 (0.05)</td>
<td>[1.09, 1.29]</td>
</tr>
</tbody>
</table>

Note. O.R.= odds ratio; SE = standard error; CI = confidence interval. Bold indicates significant effects at $p < .05$.

Figure 7: Interaction effect of presence of blindspot and cyclo-
matic complexity (CC) on puzzle accuracy. X-axis shows the
tree levels of CC: low ($\leq 2$), medium (3 – 4) and high ($> 4$).
Y-axis shows predicted accuracy (predicted probability of corre-
ctly solving a puzzle). Error bars represent 95% confidence
intervals after Bonferroni correction of the $p$-value.

that detection of security vulnerabilities benefits from a developer’s
ability and willingness to think of different scenarios and program
inputs that might cause a piece of software to generate unexpected
results. None of the other tested personality traits showed any sig-
ificant effect. This finding is in line with previous research [23]
that programming aptitude was not associated with agreeableness
or neuroticism.

4.2 Strengths and limitations

Our work takes a novel approach by analyzing blindspots in API
functions from the developers’ perspective, thereby considering vari-
bles such as perception of code, level of cognitive function, ex-
perience, and personality. This interdisciplinary approach joins
forces from computer science and psychology to understand how
API blindspots cause security vulnerabilities.

A strength of our study was that it used a behavioral approach
in addition to self-reporting by providing developers actual pro-
gramming scenarios and assessing their ability to solve them. Our
study also assessed performance-based cognitive functioning levels
as possible predictors of puzzle accuracy.

Our sample was diverse, comprising 109 developers, made up of
mostly professionals from different countries. For recruitment, we
used snowball sampling [5], which meant participants could refer
other developers. This word-of-mouth technique is often applied
in research, particularly when targeting a specific group of individ-
uals (i.e., developers). It was advantageous in allowing our team
to reach developers we could not have otherwise found using our
standard recruiting techniques (flyers, forums, social media groups,
personal networks). However, it can also introduce bias by reduc-
ing random sampling and adding possible interdependence to the
data. In our study, 41.3% of the participants chose the referral
option with only 8.3% of the referred individuals enrolling in the
study.

We conducted an a-priori power analysis to determine the appro-
propriate sample size and number of puzzles needed considering our
factorial design and with regard to our primary study aims. How-
ever, to counter possible fatigue effects, as suggested during the pi-
loting phase of this research, we asked developers to only complete
six puzzles. This resulted in a limited number of observations, thus
not allowing a robust examination of some of the effects (i.e., API
usage type, cyclo-matic complexity). Therefore, we conducted ex-
ploratory analysis on these puzzle features to generate preliminary
results, which we hope will spur future research. These prelimi-
nary results suggested that developers’ detection of blindspots was
particularly difficult for puzzles with I/O usage function and with
high cyclo-matic complexity. Increasing the number of puzzles each
developer solves would, in future research, enhance the analytic
power and allow a more comprehensive analysis of diverse puzzle
subtypes. However, to avoid fatigue and attrition, future studies
should focus on a few such categories at a time. For example,
to examine the moderation effect of I/O functions on developers’
ability to detect blindspots, I/O functions could be varied between
puzzles, while keeping cyclo-matic complexity and number of pa-
rameters consistent.

Because of compatibility issues between some developers’ browser
versions and our audio recording system software, we were not able
to collect complete cognitive data for all participants. This missing
data reduced the sample size in the analyses pertaining to the cog-
nitive measures, thus reducing power to detect significant effects.
Also, even though the cognitive tasks administered in the present
study are widely used, they may not have been sensitive enough
to differentiate between developers and/or may not have targeted
cognitive processes that are particularly relevant for detection of
blindspots in API functions.

4.3 Recommendations

Our results provide important insights for the software and API de-
velopment community and corroborates aspects of related research
in code review and developers’ perceptions of code. Our data sup-
ports the notion that blindspots in API functions lead to the intro-
duction of vulnerabilities in software, even when used by experi-
enced developers. Given these findings, API designers should con-
sider addressing developers’ misconceptions and flawed assump-
tions when working with APIs to increase code security. For ex-
ample, before release to the public, new or updated API functions
should undergo pilot testing with developers not involved in the
function’s design and implementation. This pilot testing could be
modeled after the approach used in our study. Furthermore, developer-
centric testing should be conducted with existing APIs, so that
misconceptions of specific categories of APIs can be better doc-
umented. In this context, given our preliminary findings regarding
the more pronounced effect of blindspots for I/O-related API func-

tions, greater effort should be invested in improving the design and documentation of I/O-related functions, especially considering the high prevalence of I/O operations in today’s software.

Our data did not provide support for the claim that developers’ ability to detect blindspots could be associated with their perceptions of problem difficulty, code clarity, function familiarity, confidence in their ability to solve code, their experience, expertise, and cognitive functioning, or any tested personality traits, with the exception of openness. It could be assumed that a developer who is confident and familiar with the programming scenario and API functions at hand, who has many years of programming experience, especially with a particular programming language, is cognitively high functioning, and is self-disciplined (high conscientiousness), suspicious of situations in general (low agreeableness) and emotionally stable (low neuroticism), would be better in detecting security blindspots, and would, consequently, write more secure code. These assumptions were not supported by our data. Rather, our data suggests that cognitively high functioning, experienced, confident developers can still fall for security blindspots. Software security awareness education may be a useful approach to educate developers about these risks. Such educational approaches could train developers not to rely on beliefs and gut feelings when using API functions. Increased risk awareness could lead to developers asking themselves more questions about how API function usage may result in unexpected outcomes, and could motivate them to rely more on diagnostic tools.

In large software development companies, it has become common to assign different teams to work on the various aspects of code. For example, within Google [42, 24], three distinct groups may work on functionality, security, and privacy aspects of the software separately. Such a diversified approach has the potential to minimize the introduction of vulnerabilities in code because there will be a group of developers whose primary task would be to identify how an adversary can exploit source code and cause security and privacy breaches. However, not many companies can afford to hire developers to address security alone. The common rationale is that all developers should create secure functionality. However, as discussed in Section 1, and supported by our data, this mindset maybe misleading. Both of these tasks are cognitively demanding and thus, one team to address both might be a zero-sum game.

Another practice often applied in companies is to hire an expert who is highly familiar with security vulnerabilities and has good knowledge of programming languages to decrease the chance of code vulnerabilities. Our results suggest that this rationale might also be misleading, in that even highly experienced, cognitively high functioning developers experience difficulties in detecting security blindspots in API functions.

Taken together, our study findings are applicable in the following areas: (1) design, implementation, and evaluation of new APIs; (2) addressing of blindspots in legacy APIs; (3) development of novel methods for developer recruitment/training based on personality assessment; and (4) improvement of software development processes in organizations (e.g., establishment of separate security vs. functionality teams).

5. RELATED WORK

Our work intersects the areas of API usability, programming language design, and developers’ practices and perceptions of security. In this section we provide a discussion of related work, and position our work with respect to these earlier initiatives.

5.1 API usability

Our work falls into the still young, but growing topic of API usability, which focuses on how to design APIs in a manner that reduces the likelihood of developer errors that can create software vulnerabilities. A recent article presents an overview of this field [34]. For example, Ellis et al. [17] showed that, despite its popularity, the factory design pattern [20] was detrimental to API usability because when incorporated into an API it was difficult to use.

Most studies of API usability have focused on non-security considerations, such as examining how well programmers can use the functionality that an API intends to provide. Our work is, thus, a significant departure from this research direction, although it shares many of the same methodologies.

Two of the few existing studies on security-related API usability were conducted by Coblenz et al. [10, 11] and by Weber et al. [61]. Stylos and Clarke [55] had concluded that the immutability feature of a programming language (i.e., complete restriction on an object to change its state once it is created) was detrimental to API usability. Since this perspective contradicted the standard security guidance (“Mutability, whilst appearing innocuous, can cause a surprising variety of security problems” [48, 32]), Coblenz et al. investigated the impact of immutability on API usability and security. From a series of empirical studies, they concluded that immutability had a positive effect on both security and usability [11]. Based on these findings they designed and implemented a Java language extension to realize these benefits [10].

Recent work has investigated the usability of cryptographic APIs. Nadi et al. [35] identified challenges developers face when using Java Crypto APIs, namely poor documentation, lack of cryptography knowledge by the developers, and poor API design. Acar et al. [1] conducted an online study with open source Python developers about the usability of the Python Crypto API. In this study, developers reported the need for simpler interfaces and easier-to-use code examples.

In contrast to previous work, our study focused on understanding blindspots that developers experience while working with general classes of API functions.

5.2 Programming language design

Usability in programming language design has been a long-standing concern. Initially, most of the related literature was non-empirical, but empirical studies of programming language design have become more popular. For example, Steffik and Siebert [54] showed that syntax used in a programming language was a significant barrier for novices. Our work has the potential to contribute to programming language design, since our focus is on understanding security blindspots in API function usage, and the function traits that exacerbate the problem.

5.3 Developer practices and perceptions of security and privacy

Balebako et al. discussed the relationship between the security and privacy mindsets of mobile app developers and company characteristics (e.g., company size, having a Chief Privacy Officer, etc.). They found that developers tend to prioritize security tools over privacy policies, mostly because of the language of privacy policies is so obscure [7].

Xie et al. [66] conducted interviews with professional developers to understand secure coding practices. They reported a disconnect between developers’ conceptual understanding of security and their...
attitudes regarding personal responsibility and practices for software security. Developers also often hold a “not-my-problem” attitude when it comes to securing the software they are developing; that is, they appear to rely on other processes, people, or organizations to handle software security.

Witschey et al. [63] conducted a survey with professional developers to understand factors contributing to the adoption of security tools. They found that peer effects and the frequency of interaction with security experts were more important than security education, office policy, easy-to-use tools, personal inquisitiveness, and better job performance to promote security tool adoption.

A survey conducted by Acar et al. [2] with 295 app developers concluded that developers learned security through web search and peers. The authors also conducted an experiment with over 50 Android developers to evaluate the effectiveness of different strategies to learn about app security. Programmers who used digital books achieved better security than those who used web searches. Recent research corroborates this finding by showing that the use of code-snippets from online developer forums (e.g., Stack Overflow) can lead to software vulnerabilities [3, 18, 59].

Recent studies have investigated the need and type of interventions required for developers to adopt secure software development practices. Xie et al. [65] found that developers needed to be motivated to fix software bugs. There has also been some work on how to create this motivation and encourage use of security tools. Several surveys identified the importance of social proof for developers’ adoption of security tools [33, 62, 64].

Research on the effects of external software security consultancy suggests [43] that a single time-limited involvement of developers with security awareness programs is generally ineffective in the long-term. Poller et al. [44] explored the effect of organizational practices and priorities on the adoption of developers’ secure programming. They found that security vulnerability patching is done as a stand-alone procedure, rather than being part of product feature development. In an interview-based study by Votipka et al. [60] with a group of 25 white-hat hackers and software testers on bug finding related issues, hackers were more adept and efficient in finding software vulnerabilities than testers, but they had more difficulty in communicating such issues to developers because of a lack of shared vocabulary.

In a position paper, Cappos et al. [9] proposed that software vulnerabilities are a blindspot in developers’ heuristic-based decision making mental models. Oliveira et al. [37] further showed that security is not a priority in the developers’ mindsets while coding. They found, however, that developers did adopt a security mindset once primed about the topic.

Our work complements and extends previous investigations on the effect of API blindspots on writing secure code, and in determining the extent to which developers’ characteristics (perceptions, expertise/experience, cognitive function, and personality) influence such capabilities.

6. CONCLUSIONS

In this paper, we report the results of an empirical study on understanding blindspots in API functions from the perspective of the developer. We evaluated developers’ ability to perceive blindspots in a variety of code scenarios and examined how personal characteristics, such as perceptions of the correctness of their answers, familiarity with the code, years of professional experience, level of cognitive functioning, and personality, affected this capability. We also explored the influence of programming scenario characteristics (API usage type, cyclomatic complexity) on developers’ performance in detecting blindspots.

Our study asked 109 developers to work on a set of six naturalistic programming scenarios (puzzles), comprising four puzzles with blindspots and two without blindspots. Developers were not informed about the security focus of this investigation. Our results showed that: (1) developers were less likely to correctly solve puzzles with blindspots than puzzles without blindspots, with this effect more pronounced for I/O API functions and complex code scenarios; (2) developers’ level of cognitive functioning and (3) their expertise and experience did not predict their ability to detect blindspots; however, (4) those who exhibited more openness as a personality trait did show a greater ability to detect blindspots.

Our findings have the potential to inform the design of more secure APIs. Our data suggests that API design, implementation, and testing should take into account the potential security blindspots developers may have, particularly when using I/O functions. Further, our findings that experience and cognition may not predict developers’ ability to detect blindspots, suggest that the emerging practice of establishing separate functionality vs. security teams in a given project may be a promising strategy to improve software security. This strategy may also constitute a more cost-effective paradigm for secure software development than solely relying on one group of experts, expected to simultaneously address both functionality and security.

7. ACKNOWLEDGEMENTS

We thank our shepherd Michael Reiter for guidance in writing the final version of the paper and the SOUPS 2018 anonymous reviewers for valuable feedback. We thank Sam Weber and Yanyan Zhuang for discussions related to our work. This work was supported by the National Science Foundation under grants no. CNS-1513055, CNS-1513457, and CNS-1513572.