API Blindspots: Why Experienced Developers Write Vulnerable Code

Daniela Seabra Oliveira, Tian Lin, and Muhammad Sajidur Rahman, *University of Florida;* Rad Akefirad, *Auto1 Inc.;* Donovan Ellis, Eliany Perez, and Rahul Bobhate, *University of Florida;* Lois A. DeLong and Justin Cappos, *New York University;* Yuriy Brun, *University of Massachusetts Amherst;* Natalie C. Ebner, *University of Florida*

https://www.usenix.org/conference/soups2018/presentation/oliveira

This paper is included in the Proceedings of the Fourteenth Symposium on Usable Privacy and Security.

August 12–14, 2018 • Baltimore, MD, USA

ISBN 978-1-939133-10-6

API Blindspots: Why Experienced Developers Write Vulnerable Code

Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad[α],
Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong[ν],
Justin Cappos[ν], Yuriy Brun[μ], Natalie C. Ebner
University of Florida [α]Auto1.inc [ν]New York University [μ]University of Massachusetts Amherst
daniela@ece.ufl.edu, {lintian0527, rahmanm, donovanmellis, 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 Veracode 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

USENIX Symposium on Usable Privacy and Security (SOUPS) 2018.
August 12–14, 2018, Baltimore, MD, USA.

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' understanding of (or lack thereof) the security implication of using the specific API function. After completing all puzzles, each developer provided demographic information and reported their experience and skills levels in programming languages and technical concepts. Developers then indicated endorsement of personality statements based on the Five Factor Personality Traits model [13] and completed a set of cognitive tasks from the NIH Cognition Toolbox [21] and the Brief Test of Adult Cognition by Telephone (BTACT; modified auditory version for remote use) [58].

Using quantitative statistics, we generated the following novel findings:

1. Presence of API blindspots in puzzles reduced developers' accuracy in answering the puzzles' implicit security question and reduced developers' ability to identify potential security concerns in the code.

2. API functions involving I/O were particularly likely to cause security blindspots in developers.

3. Developers were more susceptible to API blindspots for more complex puzzles, as measured by cyclomatic complexity.

4. Developers' cognitive function and their expertise and experience with programming did not predict their ability to detect API blindspots.

5. Developers exhibiting greater openness as a personality trait were more likely 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

	Professionals ($n = 70$) Mean (SD)/ %	Students ($n = 39$) Mean (SD)/ %
Gender		
Male (88)	81.4	79.5
Female (21)	18.6	20.5
Age		
Male (88)	28.0 (6.0)	24.4 (2.1)
Female (21)	27.8 (6.2)	24.4 (2.2)
Years of Programming		
	6.3 (3.5)	5.8 (5.8)
Highest Degree Earned		
High School	1.4	0.0
Some College	0.0	2.6
Associates	1.4	2.6
Bachelor's	40.0	56.4
Some Graduate School	11.4	5.1
Graduate-Level Degree	45.7	33.3
Annual Income		
0–$39,999	45.7	69.2
$40,000–$70,000	22.9	15.4
$70,001–$100,000	20.0	12.8
$100,001–$200,000	11.4	2.6
>$200,000	0.0	0.0
Race/Ethnicity		
American Indian/Alaskan	1.4	2.6
Asian	81.4	92.3
African American	2.9	0.0
Hawaiian/Pacific Islander	0.0	0.0
White	10.0	2.6
Other/Multi-racial	4.3	2.6
Country of Residence		
United States	72.3	94.9
Bangladesh	15.7	2.6
Brazil	8.6	2.6
Malaysia	1.4	0.0

Figure 1: Demographic and professional expertise/experience information about participating developers by professional group.

signed 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

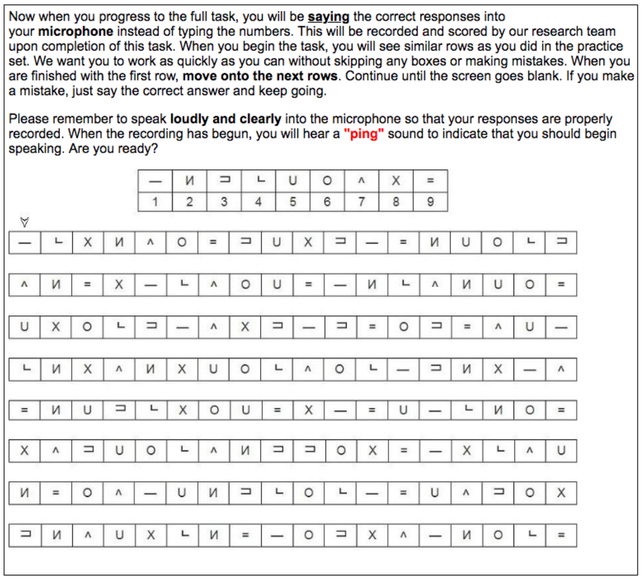

Figure 2: Oral Symbol Digit task.

the Oral Symbol Digit task we computed a sum score of correct responses. For the BTACT, we aggregated the total number of correct responses in each of the cognitive subset tasks.

2.3 Programming puzzles

This section describes the development of the programming puzzles and their characteristics. We defined a puzzle as a snippet of code simulating a real-world programming scenario. Our goal was to create concise, clear, and unambiguous puzzles that related to real-life programming tasks, while removed of code or functionality not needed for understanding the primary functionality of the code snippet. We developed two types of puzzles: those with and those without a blindspot. A blindspot puzzle targeted one particular Java API function, known to cause developers to misunderstand the security implications of its usage [40, 32]. The non-blindspot puzzles involved innocuous API functions in code context that strictly followed standard API usage protocol and code security guidelines. Puzzle development involved a two-phase, iterative process, which lasted from April 2016 to December 2016.

We chose Java as the programming language because of its rich and well-developed set of libraries and API functionalities, which can perform a diverse set of operations (including security tasks), such as I/O, multithreading, networking, random number generation, cryptography, and hashing. Java has a long-standing popularity within developer communities who use it to work on software products for different platforms, including web, mobile, and enterprise [15]. Besides being popular among professional developers, Java's wide availability of toolkits, tutorials, and online/offline resources has made it a popular choice for those learning object-oriented programming. It is the second most used programming language in GitHub repositories after Javascript [22] and was voted the third most popular technology by developers who frequently visit Stack Overflow with programming related Q&As [52]. These features made Java a good choice of programming language for our study, as we aimed to recruit from a diverse pool of developers.

Puzzle creation. This process began with a literature review to determine secure Java coding practices and the potential risks of misusing Java APIs. For puzzle selection and design principles,

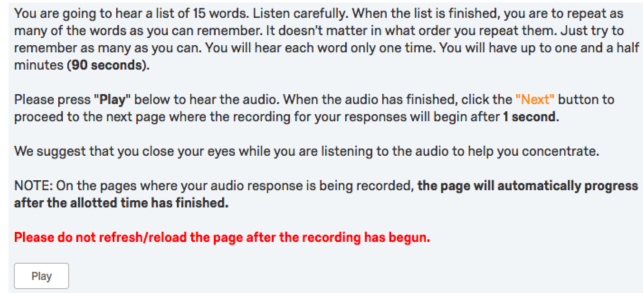

Figure 3: BTACT Word List Recall task.

we were guided by the Open Web Application Security Project (OWASP) [40], CERT's secure coding guidelines [32], vulnerability databases [36, 49], HPE's Software Security Taxonomy [19, 57], and the Java API official documentation [28]. We also leveraged programming Q&A forums, such as Stack Overflow [51] to select commonly discussed API functions. We did not to 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 different 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, including file I/O operations, garbage collection, de/serialization, cryptography, secure connection establishment, command line arguments/user inputs processing for database query, logging, user authentication, and multithreading.

Each puzzle contained four parts: (1) the puzzle scenario itself; (2) an accompanying code snippet; (3) a question about the puzzle'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 outcome 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 ensure 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 without 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`, 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:

```
1  // OMITTED: Import whatever is needed
2  public final class SystemUtils {
3    public static boolean setDate (String date)
4        throws Exception {
5      return run("DATE " + date);
6    }
7
8    private static boolean run (String cmd)
9        throws Exception {
10     Process process = Runtime.getRuntime().
        exec("CMD /C " + cmd);
11     int exit = process.waitFor();
12
13     if (exit == 0)
14       return true;
15     else
16       return false;
17   }
18 }
```

Figure 4: Sample blindspot puzzle targeting a Java Runtime API usage.

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 Java `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.**

Has Blindspot	Vulnerability (if any)	Description	API Usage Context	Targeted (non) Blindspot API function
YES	TOCTTOU race condition	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.createNewFile()` may overwrite an existing file even after the overwrite flag is set to false.	I/O	`java.io.File.createNewFile()`
YES	TOCTTOU race condition	`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.	I/O	`java.io.File.renameto()`
YES	Resurrectable object	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.	I/O	`java.lang.Object.finalize()` in the context of `java.io.File.delete()`
YES	Ambiguous return value	The `getSize()` method of the `ZipEntry` class is not reliable because it returns -1 when the size of the entry (file) is unknown. It allows an attacker to forge the field in the zip entry, which can lead to a DoS or data corruption attack.	I/O	`java.util.zip.Zipentry.getSize()`
YES	Flawed cryptographic implementation	Forgetting to call `Cipher.doFinal()` causes a Cipher object not to flush the bytes it is holding on to as the object tries to assemble a block for encrypted text. This will lead to truncated data in the final output.	Crypto	`javax.crypto.Cipher.doFinal()`
YES	Flawed cryptographic implementation	After calling `Cipher.update()`, an inappropriate selection of `Cipher.doFinal()` overloaded method (in this case, `Cipher.doFinal (byte[] input)` instead of `Cipher.doFinal(byte[] output, int outputOffset)`) creates an invalid ciphertext.	Crypto	`javax.crypto.Cipher.update()`
YES	Flawed cryptographic implementation	Failing to call `Cipher.getOutputSize()` does not guarantee the allocation of sufficient space for an output buffer, thus creating an invalid ciphertext.	Crypto	`javax.crypto.Cipher.getOutputSize()`
YES	Flawed cryptographic implementation	Failing to call `CipherOutputStream.close()` produces an invalid ciphertext, which cannot be decrypted into the original text.	Crypto	`javax.crypto.CipherOutputStream.close()`
YES	Improper input validation	Without proper input/argument sanitization, `Runtime.exec()` is vulnerable to command injection attacks.	String	`java.lang.Runtime.exec()`
YES	Improper input validation	Susceptible to inline command injection attacks without proper input sanitization.	String	`new java.lang.ProcessBuilder()`
YES	Improper input validation	Only using the `PreparedStatement` class cannot stop SQL injection attacks if string concatenation is used to build an SQL query.	String	`java.sql.PreparedStatement.setString()`
YES	Improper input validation	Inadequate input sanitization and validation allow malicious users to glean restricted information using the directory service.	String	`javax.naming.directory.DirContext.search()`
YES	Improper input validation	By using an evil regex, an attacker can make a program enter a prolonged unresponsive condition, thus enabling DoS attacks.	String	`java.util.regex.Matcher.matches()`
YES	Improper input validation	Without verifying sources, an attacker can make a program write false/unverified information into log files.	String	`java.util.logging.Logger.info()`
YES	Disclosure of sensitive information	Temporary file deletion by invoking `File.deleteOnExit()` occurs only in the case of a normal JVM shutdown, but not when the JVM crashes or is killed.	I/O	`java.io.File.deleteOnExit()`
YES	Disclosure of sensitive information	An implementation with `StandardOpenOption.DELETE_ON_CLOSE` may be unable to guarantee that it deletes the expected file when replaced by an attacker while the file is open. Consequently, sensitive data may be leaked.	I/O	`java.nio.file.Files.write()`
NO	N/A	N/A	I/O	`java.io.File.createNewFile()`
NO	N/A	N/A	I/O	`java.io.File.renameto()`
NO	N/A	N/A	I/O	`java.io.InputStream.read()`
NO	N/A	N/A	I/O	`java.util.zip.Zipentry.getSize()`
NO	N/A	N/A	I/O	`java.io.File.listFiles()`
NO	N/A	N/A	Crypto	`javax.crypto.Cipher.getOutputsize()`
NO	N/A	N/A	Crypto	`javax.crypto.CipherOutputStream.close()`
NO	N/A	N/A	String	`java.sql.PreparedStatement.setString()`

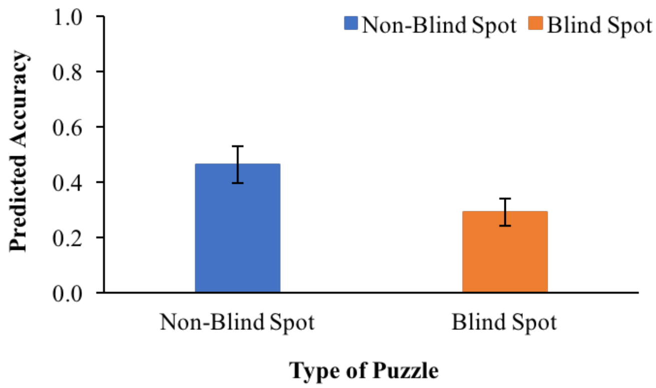

Figure 5: Developers were more likely to solve non-blindspot than blindspot puzzles. Error bars represent 95% confidence intervals.

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

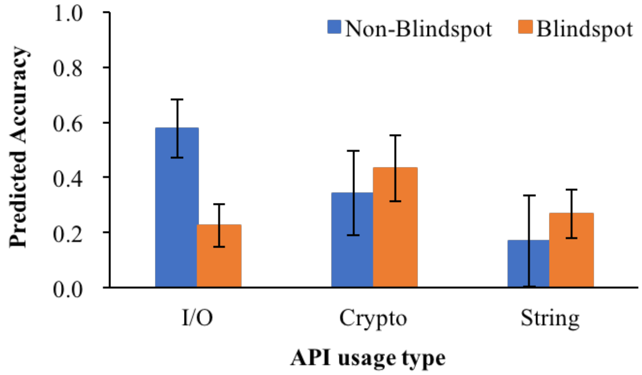

Figure 6: Interaction effect of API usage type and presence of blindspot on puzzle accuracy. The x-axis shows the three types of API usage: I/O, Crypto, and String. The y-axis shows predicted accuracy (predicted probability of correctly solving a puzzle). Error bars represent 95% confidence intervals after Bonferroni correction of the p-value.

(*Wald* $\chi^2(2) = 10.64, p = 0.005$) and its interaction with the presence of blindspot (*Wald* $\chi^2(2) = 24.81, p < 0.001$) 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).

Cyclomatic complexity. We added the categorical variable cyclomatic complexity (1 = low, 2 = medium, 3 = high) 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 puzzle cyclomatic complexity as an additional predictor. The main effect of cyclomatic complexity was not significant (*Wald* $\chi^2(1) = 0.74, p < 0.69$), but the main effect of the presence of blindspot (*Wald* $\chi^2(1) = 23.95, p < 0.001$) and its interaction with cyclomatic complexity (*Wald* $\chi^2(2) = 30.1, p < 0.001$) was significant. As shown in Figure 7, accuracy was higher for non-blindspot than for blindspot puzzles at medium cyclomatic complexity, and, even more pronounced at high cyclomatic complexity. That is, the higher the cyclomatic complexity of the code in a puzzle containing blindspots, the less likely developers were to correctly solve the puzzle.

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 multilevel regression models to examine the effect of the presence of blindspot on each of the four rating dimensions. In each model, we also considered the random effect of the intercept to accommodate for the inter-individual differences in the overall ratings for the respective dimension. As shown in Table 3, developers' perceptions did not differ as a function of the presence of blindspot in puzzles. Thus, the data did not support H2a–d.

3.3 H3–5: Cognitive function, technical expertise/experience, and personality traits

For H3, H4, and H5, the number of correctly solved blindspot puzzles across the four blindspot puzzles constituted the ordinal outcome variable blindspot puzzle accuracy, with a range from 0 to 4. Given this ordinal outcome variable, we conducted ordinal logistic regressions to test these hypotheses.

For various reasons (e.g., audio recording failure, incompatibility between the developers' browser version and our audio recording plugin and survey software), four cognitive measures from the BTACT (i.e., immediate recall, delay recall, verbal fluency, backward counting) had more than 25% data points missing. These four measures were therefore not analyzed. In addition, only 80 out of 109 developers had complete data on the Series and Digit-Back task from the BTACT, and the Oral Symbol Digit Task from the NIH toolbox. Given this missing data on the cognitive measures, which would have largely reduced the sample size, and thus power to detect significant effects, if collapsed across predictor variables (i.e., the three cognitive measures for H3 as well as the experience/expertise measures for H4 and the personality traits for H5), we conducted three separate models for H3, but tested H4 and H5 in one single model. For testing H4 and H5, three measures of professional expertise (Years of programming, Technical score, Java skills) and five personality traits (Agreeableness, Conscientiousness, Extraversion, Neuroticism, Openness) served as independent variables. As all four models (three for the cognitive measures and one for experience/expertise and personality) referred to the same dependent variable (i.e., blindspot puzzle accuracy), we applied Bonferroni correction on the threshold of the p-values ($p < 0.008$ for H3 and $p < 0.025$ for H4 and H5).

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 to: (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.

	Hypothesis 1		Expl. Anal. – API Usage Type		Expl. Anal. – Cyclomatic Complexity	
Fixed Effect	O.R. (SE)	95% CI	O.R. (SE)	95% CI	O.R. (SE)	95% CI
Presence of blindspot						
Blindspot	**0.44 (0.08)**	**[0.31, 0.63]**	**0.16 (0.05)**	**[0.09, 0.31]**	1.72 (0.55)	[0.92, 3.21]
API usage type						
Crypto			**0.33 (0.13)**	**[0.15, 0.71]**		
String			**0.11 (0.07)**	**[0.04, 0.37]**		
Presence of blindspot × API usage type						
Blindspot × Crypto			**9.10 (4.50)**	**[3.45, 23.98]**		
Blindspot × String			**11.35 (7.85)**	**[2.92, 44.04]**		
Cyclomatic complexity						
Medium					1.52 (0.68)	[0.64, 3.63]
High					**6.88 (2.62)**	**[3.26, 14.53]**
Presence of blindspot × Cyclomatic complexity						
Blindspot × Medium					**0.29 (0.15)**	**[0.10, 0.82]**
Blindspot × High					**0.02 (0.01)**	**[0.005, 0.08]**
Random Effect	σ^2 (SE)	95% CI	σ^2 (SE)	95% CI	σ^2 (SE)	95% CI
Intercept	**0.43 (0.20)**	**[0.17, 1.09]**	**0.72 (0.28)**	**[0.34, 1.52]**	**0.54 (0.25)**	**[0.22, 1.33]**
Goodness of Fit						
AIC	824.13		794.63		773.26	
BIC	837.58		826.01		804.64	

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.

	H2a: Difficulty		H2b: Clarity		H2c: Familiarity		H2d: Confidence	
Fixed Effect	B (SE)	95% CI	B (SE)	95% CI	B (SE)	95% CI	B (SE)	95% CI
Presence of Blindspot								
Blindspot	0.16 (0.14)	[-0.12, 0.43]	-0.01 (0.12)	[-0.25, 0.23]	-0.10 (0.15)	[-0.40, 0.19]	-0.11 (0.13)	[-0.36, 0.15]
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.

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

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.

Factor	Blindspot Puzzles O.R. (SE)	Blindspot Puzzles 95% CI	Non-Blindspot Puzzles O.R. (SE)	Non-Blindspot Puzzles 95% CI
Professional Expertise				
Years of programming	0.81 (0.70)	[0.15, 4.45]	3.47 (2.82)	[0.71, 17.06]
Technical expertise	0.93 (0.12)	[0.72, 1.19]	1.08 (0.12)	[0.87, 1.34]
Java skills	1.11 (0.15)	[0.85, 1.45]	0.96 (0.13)	[0.74, 1.24]
Personality Traits				
Agreeableness	0.95 (0.05)	[0.85, 1.05]	0.98 (0.04)	[0.90, 1.07]
Conscientiousness	0.97 (0.05)	[0.88, 1.07]	0.94 (0.05)	[0.85, 1.04]
Extraversion	0.94 (0.04)	[0.87, 1.01]	0.99 (0.04)	[0.91, 1.08]
Neuroticism	0.93 (0.04)	[0.86, 1.01]	**0.91 (0.04)**	**[0.83, 0.99]**
Openness	**1.18 (0.05)**	**[1.09, 1.29]**	1.08 (0.04)	[0.99, 1.17]

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

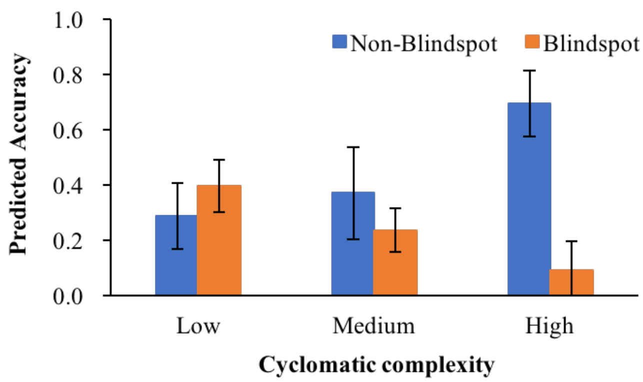

Figure 7: Interaction effect of presence of blindspot and cyclomatic complexity (CC) on puzzle accuracy. X-axis shows the three levels of CC: low (≤ 2), medium ($3 - 4$) and high (> 4). Y-axis shows predicted accuracy (predicted probability of correctly solving a puzzle). Error bars represent 95% confidence 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 significant 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 variables such as perception of code, level of cognitive function, experience, and personality. This interdisciplinary approach joints 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 programming 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 individuals (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 reducing 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 appropriate sample size and number of puzzles needed considering our factorial design and with regard to our primary study aims. However, to counter possible fatigue effects, as suggested during the piloting 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, cyclomatic complexity). Therefore, we conducted exploratory analysis on these puzzle features to generate preliminary results, which we hope will spur future research. These preliminary results suggested that developers' detection of blindspots was particularly difficult for puzzles with I/O usage function and with high cyclomatic 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 cyclomatic complexity and number of parameters 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 cognitive 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 development community and corroborates aspects of related research in code review and developers' perceptions of code. Our data supports the notion that blindspots in API functions lead to the introduction of vulnerabilities in software, even when used by experienced developers. Given these findings, API designers should consider addressing developers' misconceptions and flawed assumptions when working with APIs to increase code security. For example, 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 documented. In this context, given our preliminary findings regarding the more pronounced effect of blindspots for I/O-related API functions, 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 positive effects 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-consult documentation with secure, easy-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, Stefik 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.

Acar et al. [4] and Green and Smith [27] suggest a research agenda to achieve usable security for developers. They proposed several research questions to elicit developers' attitudes, needs, and priorities in the area of security. Oltrogge et al. [38] asked for developers' feedback on TLS certificate pinning strategy in non-browser based mobile applications. They found a wide conceptual gap about pinning and its proper implementation in software due to API complexity.

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.

8. REFERENCES

[1] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. Comparing the Usability of Cryptographic APIs. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 154–171, May 2017.

[2] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 289–305, May 2016.

[3] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. How Internet Resources Might Be Helping You Develop Faster but Less Securely. *IEEE Security Privacy*, 15(2):50–60, March 2017.

[4] Y. Acar, S. Fahl, and M. L. Mazurek. You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 3–8, Nov 2016.

[5] R. Atkinson and J. Flint. Accessing Hidden and Hard-to-Reach Populations: Snowball Research Strategies. *Social research update*, 33(1):1–4, 2001.

[6] J. Bailey and R. B. Mitchell. Industry Perceptions of the Competencies Needed by Computer Programmers: Technical, Business, and Soft Skills. *Journal of Computer Information Systems*, 47(2):28–33, 2006.

[7] R. Balebako, A. Marsh, J. Lin, J. I. Hong, and L. F. Cranor. The Privacy and Security Behaviors of Smartphone App Developers. In *Proceedings of 2014 Workshop on Usable Security*. Internet Society, 2014.

[8] K. P. Burnham and D. R. Anderson. Multimodel Inference: Understanding AIC and BIC in Model Selection. *Sociological methods & research*, 33(2):261–304, 2004.

[9] J. Cappos, Y. Zhuang, D. Oliveira, M. Rosenthal, and K.-C. Yeh. Vulnerabilities As Blind Spots in Developer's Heuristic-Based Decision-Making Processes. In *Proceedings of the 2014 New Security Paradigms Workshop*, NSPW '14, pages 53–62, New York, NY, USA, 2014. ACM.

[10] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine. Glacier: Transitive Class Immutability for Java. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 496–506, Piscataway, NJ, USA, 2017. IEEE Press.

[11] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull. Exploring Language Support for Immutability. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 736–747, New York, NY, USA, 2016. ACM.

[12] Common Weakness Enumeration (CWE)/SANS Top 25 Most Dangerous Software Errors, 2011. Available at `http://cwe.mitre.org/top25/`.

[13] P. T. Costa and R. R. MacCrae. *Revised NEO Personality Inventory (NEO PI-R) and NEO Five-Factor Inventory (NEO-FFI): Professional Manual*. Psychological Assessment Resources, Incorporated, 1992.

[14] B. Dagenais and M. P. Robillard. Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 127–136, New York, NY, USA, 2010. ACM.

[15] N. Diakopoulos and S. Cass. The Top Programming Languages 2016, Jul 2016. Available at `https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016`.

[16] A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner. An Empirical Study on the Effectiveness of Security Code Review. In *Proceedings of the 5th International Conference on Engineering Secure Software and Systems*, ESSoS'13, pages 197–212, Berlin, Heidelberg, 2013. Springer-Verlag.

[17] B. Ellis, J. Stylos, and B. Myers. The Factory Pattern in API Design: A Usability Evaluation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 302–312, Washington, DC, USA, 2007. IEEE Computer Society.

[18] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack Overflow Considered Harmful? The Impact of Copy Paste on Android Application Security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136, May 2017.

[19] A Taxonomy of Coding Errors that Affect Security. Available at `https://vulncat.hpefod.com/en`.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[21] R. C. Gershon, M. V. Wagster, H. C. Hendrie, N. A. Fox, K. F. Cook, and C. J. Nowinski. NIH Toolbox for Assessment of Neurological and Behavioral Function. *Neurology*, 80(11 Supplement 3):S2–S6, 2013.

[22] GitHut : Discover Languages in Github, 2014. Available at `http://githut.info/`.

[23] T. Gnambs. What Makes a Computer Wiz? Linking Personality Traits and Programming Aptitude. *Journal of Research in Personality*, 58:31 – 34, 2015.

[24] Google's Approach to IT Security: A Google White Paper, 2016. Available at `https://static.googleusercontent.com/media/1.9.22.221/en//enterprise/pdf/whygoogle/google-common-security-whitepaper.pdf`.

[25] D. Gopstein, J. Iannacone, Y. Yan, L. DeLong, Y. Zhuang, M. K.-C. Yeh, and J. Cappos. Understanding Misunderstandings in Source Code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 129–139, New York, NY, USA, 2017. ACM.

[26] GraphicsMagick 1.4 Heap-Based Buffer Overflow Vulnerability, Dec. 2017. Available at `https://nvd.nist.gov/vuln/detail/CVE-2017-17915`.

[27] M. Green and M. Smith. Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security Privacy*, 14(5):40–46, Sept 2016.

[28] Java Platform SE 8 Documentation. Available at `https://docs.oracle.com/javase/8/docs/`.

[29] O. P. John and S. Srivastava. The Big Five Trait Taxonomy: History, Measurement, and Theoretical Perspectives. *Handbook of personality: Theory and research*, 2(1999):102–138, 1999.

[30] P. J. Kovacs and G. A. Davis. Determining Critical Skills and Knowledge Requirements of It Professionals by Analysing Keywords in Job Posting. In *48th Annual IACIS International Conference*. IACIS, 2008.

[31] T. J. McCabe. A Complexity Measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976.

[32] J. McManus and S. Shrum. SEI CERT Oracle Coding Standard for Java. Available at `https://wiki.sei.cmu.edu/confluence/display/java/Java Coding Guidelines`.

[33] E. Murphy-Hill, D. Y. Lee, G. C. Murphy, and J. McGrenere. How Do Users Discover New Tools in Software Development and Beyond? *Computer Supported Cooperative Work (CSCW)*, 24(5):389–422, Oct 2015.

[34] B. A. Myers and J. Stylos. Improving API Usability. *Commun. ACM*, 59(6):62–69, May 2016.

[35] S. Nadi, S. Krüger, M. Mezini, and E. Bodden. Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 935–946, New York, NY, USA, 2016. ACM.

[36] National Vulnerability Database. Available at `https://nvd.nist.gov/`.

[37] D. Oliveira, M. Rosenthal, N. Morin, K.-C. Yeh, J. Cappos, and Y. Zhuang. 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*, ACSAC '14, pages 296–305, New York, NY, USA, 2014. ACM.

[38] M. Oltrogge, Y. Acar, S. Dechand, M. Smith, and S. Fahl. To Pin or Not to Pin Helping App Developers Bullet Proof Their TLS Connections. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 239–254, Berkeley, CA, USA, 2015. USENIX Association.

[39] The Open Web Application Security Project (OWASP) Top 10 Most Critical Web Application Security Risks, 2013. Available at `https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf`.

[40] OWASP Secure Coding Practices Checklist, 2016. Available at `https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_Checklist`.

[41] H. Orman. The Morris Worm: a Fifteen-year Perspective. *IEEE Security Privacy*, 1(5):35–43, Sept 2003.

[42] Personal communication with a Google project team leader.

[43] A. Poller, L. Kocksch, K. Kinder-Kurlanda, and F. A. Epp. First-time Security Audits As a Turning Point?: Challenges for Security Practices in an Industry Software Development Team. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '16, pages 1288–1294, New York, NY, USA, 2016. ACM.

[44] A. Poller, L. Kocksch, S. Türpe, F. A. Epp, and K. Kinder-Kurlanda. Can Security Become a Routine?: A Study of Organizational Change in an Agile Software Development Group. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, CSCW '17, pages 2489–2503, New York, NY, USA, 2017. ACM.

[45] M. S. Rahman. An Empirical Case Study on Stack Overflow to Explore Developers' Security Challenges. Masters Report, Available at `http://krex.k-state.edu/dspace/handle/2097/34563`, 2016.

[46] M. P. Robillard. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, 26(6):27–34, Nov 2009.

[47] M. P. Robillard and R. Deline. A Field Study of API Learning Obstacles. *Empirical Softw. Engg.*, 16(6):703–732, Dec. 2011.

[48] Secure Coding Guidelines for Java SE. Available at `http://www.oracle.com/technetwork/java/seccodeguide-139067.html#6`.

[49] Security Focus Vulnerability Database. Available at `https://www.securityfocus.com/`.

[50] Security Vulnerabilities Published In 2017 (SQL Injection), 2017. Available at `https://www.cvedetails.com/vulnerability-list/opsqli-1/sql-injection.html`.

[51] Stack Overflow: A Q/A Site for Professional and Enthusiast Programmers. Available at `https://www.stackoverflow.com/`.

[52] Stack Overflow Developer Survey, 2016. Available at `https://insights.stackoverflow.com/survey/2016`.

[53] State of Software Security, 2016. Available at `https://www.veracode.com/sites/default/files/Resources/Reports/state-of-software-security-volume-7-veracode-report.pdf`.

[54] A. Stefik and S. Siebert. An Empirical Investigation into Programming Language Syntax. *Trans. Comput. Educ.*, 13(4):19:1–19:40, Nov. 2013.

[55] J. Stylos and S. Clarke. Usability Implications of Requiring Parameters in Objects' Constructors. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 529–539, Washington, DC, USA, 2007. IEEE Computer Society.

[56] Symantec Internet Security Threat Report, 2017. Available at `https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf`.

[57] K. Tsipenyuk, B. Chess, and G. McGraw. Seven Pernicious Kingdoms: a Taxonomy of Software Security Errors. *IEEE Security Privacy*, 3(6):81–84, Nov 2005.

[58] P. A. Tun and M. E. Lachman. Telephone Assessment of Cognitive Function in Adulthood: the Brief Test of Adult Cognition by Telephone. *Age and Ageing*, 35(6):629–632, 2006.

[59] T. Unruh, B. Shastry, M. Skoruppa, F. Maggi, K. Rieck, J.-P. Seifert, and F. Yamaguchi. Leveraging Flawed Tutorials for Seeding Large-Scale Web Vulnerability Discovery. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association.

[60] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek. Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 134–151, 2018.

[61] S. Weber, M. Coblenz, B. Myers, J. Aldrich, and J. Sunshine. Empirical Studies on the Security and Usability Impact of Immutability. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 50–53, Sept 2017.

[62] J. Witschey, S. Xiao, and E. Murphy-Hill. Technical and Personal Factors Influencing Developers' Adoption of Security Tools. In *Proceedings of the 2014 ACM Workshop on Security Information Workers*, SIW '14, pages 23–26, New York, NY, USA, 2014. ACM.

[63] J. Witschey, O. Zielinska, A. Welk, E. Murphy-Hill, C. Mayhorn, and T. Zimmermann. Quantifying Developers' Adoption of Security Tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 260–271, New York, NY, USA, 2015. ACM.

[64] S. Xiao, J. Witschey, and E. Murphy-Hill. Social Influences on Secure Development Tool Adoption: Why Security Tools Spread. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '14, pages 1095–1106, New York, NY, USA, 2014. ACM.

[65] J. Xie, H. Lipford, and B.-T. Chu. Evaluating Interactive Support for Secure Programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2707–2716, New York, NY, USA, 2012. ACM.

[66] J. Xie, H. R. Lipford, and B. Chu. Why do Programmers Make Security Errors? In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 161–164. IEEE, 2011.