Blindspots in Python and Java APIs Result in Vulnerable Code

YURIY BRUN, University of Massachusetts Amherst
TIAN LIN and JESSIE ELISE SOMERVILLE, University of Florida
ELISHA M. MYERS, Florida Atlantic University, University of Florida
NATALIE EBNER, University of Florida

Blindspots in APIs can cause software engineers to introduce vulnerabilities, but such blindspots are, unfortunately, common. We study the effect APIs with blindspots have on developers in two languages by replicating a 109-developer, 24-Java-API controlled experiment. Our replication applies to Python and involves 129 new developers and 22 new APIs. We find that using APIs with blindspots statistically significantly reduces the developers' ability to correctly reason about the APIs in both languages, but that the effect is more pronounced for Python. Interestingly, for Java, the effect increased with complexity of the code relying on the API, whereas for Python, the opposite was true. This suggests that Python developers are less likely to notice potential for vulnerabilities in complex code than in simple code, whereas Java developers are more likely to recognize the extra complexity and apply more care, but are more careless with simple code. Whether the developers considered API uses to be more difficult, less clear, and less familiar did not have an effect on their ability to correctly reason about them. Developers with better long-term memory recall were more likely to correctly reason about APIs with blindspots, but short-term memory, processing speed, episodic memory, and memory span had no effect. Surprisingly, professional experience and expertise did not improve the developers' ability to reason about APIs with blindspots across both languages, with long-term professionals with many years of experience making mistakes as often as relative novices. Finally, personality traits did not significantly affect the Python developers' ability to reason about APIs with blindspots, but less extroverted and more open developers were better at reasoning about Java APIs with blindspots. Overall, our findings suggest that blindspots in APIs are a serious problem across languages, and that experience and education alone do not overcome that problem, suggesting that tools are needed to help developers recognize blindspots in APIs as they write code that uses those APIs.

CCS Concepts: • **Software and its engineering**; • **Human-centered computing → User studies**; • **Human computer interaction → Software security engineering**;

Additional Key Words and Phrases: Software vulnerabilities, Java, Python, APIs, API blindspots

This work is supported by the National Science Foundation under grants CCF-1453474, CNS-1513055, CNS-1513457, CNS-1513572, and CCF-1564162.

Authors' addresses: Y. Brun, University of Massachusetts Amherst, 140 Governors Drive, Amherst, MA, 01003-9264; email: brun@cs.umass.edu; T. Lin, J. E. Somerville, and N. Ebner, University of Florida, Gainesville, FL, 32611-2250; emails: lintian0527@ufl.edu, JSomerville@dental.ufl.edu, natalie.ebner@ufl.edu; E. M. Myers, Charles E. Schmidt College of Medicine, Florida Atlantic University, Boca Raton, FL, 33431 and University of Florida, Gainesville, FL, 32611-2250; email: myerse2020@health.fau.edu.

1049-331X/2023/04-ART76 $15.00
https://doi.org/10.1145/3571850

ACM Reference format:
Yuriy Brun, Tian Lin, Jessie Elise Somerville, Elisha M. Myers, and Natalie Ebner. 2023. Blindspots in Python and Java APIs Result in Vulnerable Code. *ACM Trans. Softw. Eng. Methodol.* 32, 3, Article 76 (April 2023), 31 pages.
https://doi.org/10.1145/3571850

1 INTRODUCTION

Vulnerabilities are, unfortunately, ubiquitous in modern software [94]. For example, in 2006, Mozilla had 300 bugs reported per day [7], and per bugzilla.mozilla.org, the situation has not improved since. In 2013, the global cost of debugging was $312 billion [18]. In 2020, the global cost of poor quality in legacy systems was $520 billion, and the cost of operational software failures $1.56 trillion [58]. The fact that programs ship with both known and unknown bugs is a well-known and accepted fact [61]. Web-based software, in particular, suffers from quality problems, with 76% of all websites containing software vulnerabilities; 9% of them critical [94].

It is important to note that most vulnerabilities are not from new causes. New applications often contain instances of vulnerabilities that have been known for years: 61% of all web applications contain one of the vulnerabilities captured by the OWASP Top 10 2013 vulnerability categories list [73], such as information leakage, flawed cryptographic implementations, and carriage-return-line-feed (CRLF) injection [91]. And 66% of the vulnerabilities represent programming practices that fail to avoid the top 25 most dangerous programming errors [31]. New instances of existing, well-known vulnerabilities, such as SQL injections and buffer overflows, are frequently reported in new software [48, 88].

One way in which vulnerabilities find their way into systems is when developers use application programming interfaces (APIs) in unsafe or unintended ways. The problem lies, in part, in that APIs can be counterintuitive. For example, use of `strcpy()`—known for nearly three decades [74] to lead to a buffer overflow vulnerability if developers do not check and match sizes of the destination and source arrays—can often cause *blindspots* in a developer's mind. As a developer put it, *"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"* [70]. Using APIs can be difficult. Mapping requirements to proper API usage protocols, understanding API side effects, and even deciding between differing expert opinions on API use all pose challenges [47, 79, 80]. Developers misunderstanding APIs is frequently the cause of security vulnerabilities [28, 33, 78].

So, why do developers continue to use APIs that are known to cause vulnerabilities? Developers often blindly trust APIs, which can lead to blindspots: misconceptions, misunderstandings, or oversights in how APIs are expected to be used [28]. Developers can even feel that they're outsourcing the responsibility for ensuring security when using APIs, not feeling themselves responsible for ensuring the API does the right thing with respect to security [70]. These blindspots can lead to violations of the recommended API usage protocols and to the introduction of security vulnerabilities, if, for example, API functions invocations have security implications that are not readily apparent to the developer.

To study how blindspots affect developers, we recently designed and executed a Java-based controlled study with 109 developers working on programming tasks called puzzles, which involve answering questions about the expected behavior of small programs that use APIs [71]. Unlike many studies, more than 70% of our participants were professional developers (and less than 30% students); the average developer had more than six years of programming experience. We found that (1) the presence of APIs with known blindspots reduced developers' accuracy in answering

security questions and their ability to identify potential security concerns in the code; (2) more complex code puzzles, as measured by cyclomatic complexity, led to more developer confusion; (3) surprisingly, developers' cognitive function and expertise and experience did not predict their ability to detect API blindspots, but (4) developers exhibiting greater openness and lower extraversion were more likely to detect API blindspots. These findings have allowed research into understanding why developers make security mistakes [75, 101], gaining insight into the developers' rationale in making API-use decisions [100], and evaluating the usability of security APIs [105] including their application to smart contracts [103]. Unfortunately, our prior study had several limitations. Most notably, it was limited to Java development. As such, while the findings shed some light on challenges that arise in using APIs, the study could not determine which findings generalize across languages and which are specific to Java, or, in fact, only to the API blindspots tested.

To address this shortcoming, in this article, we replicate that study with 129 new developers, and 22 new puzzles incorporating all new APIs, 16 of which contain known blindspots. Most importantly, our replication study is entirely in Python, allowing us to identify observations that are language-specific versus those that generalize to broader development practices. Our study's 129 developers include professional developers and senior undergraduate and graduate students (91 professionals, 38 students, 28.6 years old on average, 89.9% male). We designed each puzzle to contain a short code snippet simulating a real-world programming scenario. Of these puzzles, 16 contained one API function known to cause developers to experience blindspots; we developed these puzzles based on API functions commonly reported in vulnerability databases [68, 87] or frequently discussed in developer fora [90]. The other 6 puzzles involved an innocuous API function. The API functions could each be classified into three categories: input/output (I/O), cryptography, and string manipulation. Following the 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 [32] and completed a set of cognitive tasks from the NIH Cognition Toolbox [44] and the Brief Test of Adult Cognition by Telephone (BTACT; modified auditory version for remote use) [98].

Our study, together with the data from our prior, Java-based study [71], supports the following conclusions:

- Developers are statistically significantly less likely to solve puzzles with APIs with blindspots than those without blindspots. The effect is more pronounced for Python than Java.
- The complexity of the puzzle had a different effect on the Python and Java developers. For Python, developers were far more often correct when solving low-complexity puzzles with APIs without blindspots than with blindspots, but there was virtually no difference for high-complexity puzzles. For Java, the opposite was true, with developers far more often correct when solving high-complexity puzzles with APIs without blindspots than with blindspots.
- Whether the developers considered the puzzles more difficult, less clear, and less familiar did not affect their ability to solve the puzzles.
- Developers with better long-term memory recall were more likely to correctly solve puzzles with blindspots, but short-term memory, processing speed, episodic memory, and memory span had no effect.
- Surprisingly, professional experience and expertise did not improve the developers' ability to solve puzzles with APIs with blindspots across both languages.

- Developers with lower extraversion and higher openness as personality traits were more accurate in solving Java puzzles with APIs with blindspots, but for Python, developers' personality traits were not associated with their ability to solve such puzzles.

We make public all Java and Python puzzles for use in future research as well as the surveys, data, and analysis code here: https://osf.io/ahpfv/?view_only=37978cb1e67941d5b1e572f2fba982c9.

As our study improves the understanding of how blindspots affect developers, we aim to inform subsequent research for improving software quality, e.g., via the design of tools that help alert the developers to potential vulnerabilities in their code when using APIs with blindspots, via managerial decisions such as who is assigned to write or review certain code, or via improved educational methods to reduce the negative effects of blindspots in APIs on software quality. For example, our study finds that professional experience and expertice did not, alone, improve the developers' ability to reason about APIs, suggesting that simply ensuring at least one experienced developer reviews all production code is an insufficient mitigation strategy. Our observation that Python developers made frequent mistakes in low-complexity puzzles with blindspots suggests that the developers may apply less care to seemingly simple coding scenarios, and that tools that warn developers of potential dangers and refocus their attention may help. Our results can inform future creation of such tools and methods for improving software quality.

The remainder of this article is organized as follows: Section 2 describes the API puzzles. Section 3 presents our study's methodology. Section 4 presents our experimental design and assesses the results, and Section 5 discusses some of the implications of these findings. Section 6 places this study in the context of related work, and Section 7 summarizes our contributions.

2 API PUZZLES

Our study is focused around puzzles, snippets of code that simulate a real-world programming scenario. The snippets of code use APIs, some of which are known to contain blindspots.

The goal of a puzzle is to simulate a clear, concise, and unambiguous programming task representative of a real-world programming task. Users solving puzzles would necessarily interact with the contained APIs. As some of the APIs contain known blindspots, while others do not, we aim to use the users' attempts to solve the puzzles to understand the impact of blindspots on their behavior. Our study counterbalanced puzzle selection so each participant received 6 randomly selected puzzles out of a pool of 22, with 4 of those puzzles containing a blindspot (details described in Section 3).

Figure 1 shows a sample Python puzzle. This code in the puzzle uses the `input` API, which has a known blindspot. The fact that `print` on lines 2, 6, and 8 are statements (not functions), implies that this code is written in Python version 2.x. The `input` function executes the input typed by the user. If the input the user types is a Python expression, then Python will execute that expression. This means that the user can type arbitrary code and the program will execute it, potentially corrupting data or giving up control of the machine—a serious, known vulnerability called a *code injection* attack.[1]

Each puzzle, as the one in Figure 1, consists of three elements: the scenario description, the code, and two questions.

We collected API functions commonly reported in vulnerability databases [68, 87] or frequently discussed in developer fora [90] and used these APIs to create puzzles, small programs that exercise the APIs. Each program aimed to miminize its size and cyclomatic complexity while exercising the behavior of the API necessary to expose the potentially unexpected behavior of the blindspot

[1] https://www.cvedetails.com/cve/CVE-2018-1000802/.

Scenario:
Consider the following snippet of Python code that searches for a particular file in a file system, for which a path is entered by the user. The code uses the `os.path.exists(path)` function, which returns **`true`** if the string passed as the parameter refers to an existing path, and **`false`** otherwise. Consider the snippet of code below and answer the following questions, assuming that the code has all required permissions to execute.

```
1 import os
2 print "For which pathname do you want to search?"
3 # The user enters a file pathname to search
4 path = input('> ')
5 if os.path.exists(str(path)):
6   print path, "pathname exists."
7 else:
8   print path, "pathname does not exist."
```

Questions:

(1) What will the program do when executed?
(2) What will happen if the user enters the string "`os.system("date")`" when prompted for the file pathname?
  (a) The current date will be displayed on the terminal.
  (b) The program will print "pathname does not exist."
  (c) The program will crash with error message "invalid input".
  (d) The program will crash with no error message.
  (e) None of the above.

Fig. 1. A sample Python puzzle (Puzzle 1 in Figure 3). The `input` API, in Python 2.7, contains a known blindspot: The user's unfiltered input can be executed directly by the program using the `os` package. The correct answer to question 1 is that the program first reads a line from the standard input and evaluates it as a Python expression, and then prints the result of that evaluation to the screen, followed by either "pathname exists." or "pathname does not exist." depending on whether a file with the evaluated expression's name exists in the current path. Because of the blindspot in the `input` API, the user can use the `os` package to execute arbitrary Python code. So, the correct answer to question 2 is "a."

or the innocuous API. We created a total of 22 Python puzzles; 16 contained APIs with blindspots and 6 contained APIs without blindspots using innocuous API functions not known to cause vulnerabilities. Figure 2 shows an example puzzle that uses an API without a known blindspot.

Figure 3 summarizes the 22 puzzles we developed for our study. These puzzles are all available in our data package: https://osf.io/ahpfv/?view_only=37978cb1e67941d5b1e572f2fba982c9.

The APIs used in the puzzles come from three categories: input/output (I/O), cryptography, and string manipulation. I/O APIs involve operations such as networking activity, and reading and writing from and to streams, files, and internal memory buffers. Cryptography APIs include encryption, decryption, and key agreement. String manipulation APIs include editing and processing strings, such as queries and user input.

We designed the puzzles aiming to keep each puzzle's complexity to the minimum necessary to properly capture the API's use. As a result, the complexity of the puzzles varied. We measured the puzzles complexity using cyclomatic complexity, a quantitative measure of the number of linearly independent paths in the source code [62]. We classified the puzzles as *low* complexity (cyclomatic complexity of 1–2), *medium* complexity (3–4), and high complexity (≥5). Of the 22 puzzles, 12 were low complexity, 7 medium complexity, and 3 high complexity.

Our earlier, Java-based study [71] used a total of 24 puzzles, written in Java (16 of which used APIs with blindspots and 8 used innocuous APIs). Figure 4 shows a sample Java puzzle with an API with a blindspot. This puzzle uses the `Runtime.exec` API, which executes code passed to it via its argument, also exposing the `setDate` method to a code injection attack.

All Java and Python puzzles, as well as the surveys, data, and analysis code, are available here: https://osf.io/ahpfv/?view_only=37978cb1e67941d5b1e572f2fba982c9.

Scenario:
Consider this snippet of Python code that implements a music directory by storing information about song titles, artist names, album names, and release dates in a MySQL database. The user is asked to enter the genre of music for which they are searching. This input is then passed to a database, which uses this information to select songs and returns them to the user. Consider the snippet of code below and answer the following questions, assuming that the code has all required permissions to execute.

```
# Import whatever is needed
genre = input("Please enter your desired genre>")
# Open database connection
db = MySQLdb.connect("localhost","allUsers","54373$!@","MUSICDB")
'''
Prepares a cursor object that will hold the
information on the rows accessed by our query
'''
cursor = db.cursor()
cursor.execute("SELECT songName, albumName, releaseDate "
    "FROM Songs WHERE genre = '%s'", genre)
# Get all the rows
results = cursor.fetchall()
for row in results:
   print(row)
db.close()
```

Questions:

(1) What will the program do when executed?
(2) What type of information could a user retrieve from this program?
  (a) The song name
  (b) The album name
  (c) The artist name
  (d) A and B
  (e) Any attribute from the database

Fig. 2. A sample Python puzzle (Puzzle 20 in Figure 3) that uses the `cursor.execute` API that does not contain a known blindspot. The correct answer to question 1 is that the function will return records from the database that match the specified genre. The correct answer to question 2 is "d."

3 DATA-COLLECTION METHODOLOGY

We first describe our study's participants (Section 3.1) and then our data-collection procedure (Section 3.2). Our study protocol involved human subjects and was approved by the University of Florida Institutional Review Board (protocol #IRB201701817). Once the data were collected and anonymized, researchers performed secondary analysis at both the University of Florida and the University of Massachusetts Amherst, following both institutions' Institutional Review Board recommendations.

3.1 Participants

Our study targeted developers who actively use the Python programming language. We recruited 129 experienced Python developers. We considered individuals with more than one year of experience in a professional setting to be *professionals*; we considered all others to be *students*. Collectively, we refer to all participants as *developers* in this study. Recruitment methods included flyers and handouts, social media advertisements, advertisements through companies and their human resources departments, university listservs, and contacts via the authors' personal networks, as well as word-of-mouth. Professionals were compensated $50 and students $20 for study completion. These different compensation amounts were based on level of programming experience and were approved by the Institutional Review Board; they were justified given a larger financial incentive necessary to recruit professional developers, in consideration of their relatively high-paying jobs and a more limited availability.

id	API	type	cyclomatic complexity	blindspot description
1	`input`	I/O	2	In Python v2.7, passing a user-specified string to `input` allows execution of arbitrary code.
2	`eval`	string	5	Passing a user-supplied string to `eval` allows execution of arbitrary code.
3	`MySQLdb`'s `cursor.execute`	string	2	Passing a user-supplied string to `MySQLdb`'s `cursor.execute` allos retrieving arbitrary data from the database.
4	`os.path.join`	I/O	3	Without sanitizing the input, `os.path.join` allows the user to access to all files on the filesystem.
5	`subprocess.call`	string	1	Passing a user-supplied string to `subprocess.call` allows execution of arbitrary code.
6	`pickle.loads`	string	3	The `pickle.loads` function executes data received from the client as a shell command, allowing execution of arbitrary code.
7	`ssl.SSLContext`	crypto	2	Without explicitly setting the context for `ssl.SSLContext`, the default configuration imposes no authentication.
8	`xml.sax.parse`	string	2	Maliciously designed data can cause `xml.sax.parse` to run for time exponential in the size of the data, allowing for denial of service attacks.
9	`hash`	crypto	3	An improperly implemented `hash` function can lead identical document to appear to be different.
10	`os.tmpnam`	I/O	2	A limit on the number of distinct temporary filenames `os.tmpnam` generates can cause a reuse of a filename and data loss.
11	`shutil.copy`	I/O	3	The `shutil.copy` function loses file metadata when copying files.
12	`os.access`	I/O	3	The accessibility of a file can change between when `os.access` checks file accessibility and when the file is accesssed.
13	`os.walk`	I/O	3	Setting `followlinks` to `True` in `os.walk` can lead to infinite recursion if the current directory contains a link to a parent directory.
14	`compare_digest`	crypto	5	The `compare_digest` function is vulnerable to timing attacks; the time comparing two strings takes correlates with the location of the first difference.
15	`TarFile.extract`	I/O	4	The `TarFile.extract` function can extract files outside of the current location if those files are encoded using absolute locations.
16	`curs.execute`	string	2	The `curs.execute` function is vulnerable to SQL injection attacks.
17	`raw_input`	I/O	1	No blindspot exposed.
18	`cursor.execute`	string	2	No blindspot exposed.
19	`ssl.SSLContext`	crypto	2	No blindspot exposed.
20	`compare_digest`	crypto	5	No blindspot exposed.
21	`os.fwalk`	I/O	2	No blindspot exposed.
22	`curs.execute`	string	2	No blindspot exposed.

Fig. 3. The 22 Python puzzles used in our study.

Initially, we received a total of 423 emails from interested developers. Out of the 423, 13 (3.1%) participants explicitly dropped out, and 184 (43.5%) participants implicitly dropped out by becoming unresponsive at some point in the data-collection process. The remaining 226 developers (53.4%) completed the study. Of those 226 developers, 84 (37.2%) developers' data was incomplete, e.g., they did not answer some of their puzzles' questions, did not complete the audio task for cognitive assessment (described in Section 4.4), or had technical difficulties, such as browser

Scenario:
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.

```
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 }
```

Questions:

(1) What will the `setDate` method do when executed?
(2) If a program calls the `setDate` method with an arbitrary `String` value as the new date, which one statement is correct?
  (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 method might do more than changing the date.
  (d) The return value of the `waitFor` method is not interpreted correctly (lines 14 17).
  (e) The web application will crash.

Fig. 4. A sample Java puzzle. This puzzle uses the `Runtime.exec` API, which executes code passed to it via its argument. Because the `run` method, called by the `setDate` method, simply passes `setDate`'s arbitrary `String` argument to `Runtime.exec`, this method may end up executing arbitrary code. The correct answer to question 2 is "c."

incompatibility issues, that prevented audio recording. We discarded these 84 developers' data. During the enrollment process, we detected that 13 (5.6%) of the developers were repeat participants; that is, they participated multiple times despite instructions not to do so. For those individuals, we accepted their first data submission as a valid entry and discarded all subsequent submissions. That resulted in 129 valid developer sessions. Unless otherwise stated, we report our results based on that sample of 129 developers who proceeded through all study procedures as instructed and completed all tasks.

Figure 5 summarizes the participant demographics and their professional expertise and experience. The 129 developers consisted of 91 (70.5%) professional developers and 38 (29.5%) students. The vast majority of developers (125, 96.9%) had 2 or more years of Python programming experience. Student participants self-reported a relatively high programming experience (mean of 5.3 years, standard deviation of 2.9 years), likely a result of programming prior to entering the university or being students for more than six years (e.g., PhD students). Participants ranged between the ages of 18 and 71 years (mean of 28.4 years, standard deviation of 7.8 years), with the large majority of participants being male (116, 89.9%). Participants were recruited across the globe with concentrations in North and South America (74, 57.4%), Asia (43, 33.3%), and Europe (11, 8.7%). Overall, participants came from the United States, Brazil, India, Bangladesh, Pakistan, Greece, Poland, France, United Kingdom, Germany, and so on.

	Professionals	Students
Total	91	38
Years of Programming	6.9 (stdev 5.6)	4.1 (stdev 2.1)
Age	30.1 (stdev 7.2)	25.1 (stdev 4.4)
Gender		
Male (116)	82 (90.1%)	34 (89.5%)
Female (13)	9 (9.9%)	4 (10.5%)
Highest Degree Earned		
High School or GED	2 (2.2%)	3 (7.9%)
Some College	1 (1.1%)	3 (7.9%)
Associates / 2-year degree	1 (1.1%)	1 (2.6%)
Bachelor's / 4-year degree	41 (45.1%)	16 (42.1%)
Some Graduate School	5 (5.5%)	5 (13.1%)
Graduate-Level Degree	41 (45.1%)	10 (26.3%)
Annual Income		
$0– $39,999	43 (47.2%)	32 (86.8%)
$40,000– $70,000	20 (22.0%)	4 (10.8%)
$70,000–$100,000	15 (16.5%)	0 (0.0%)
$100,001–$200,000	11 (12.1%)	0 (0.0%)
$201,000+	2 (2.2%)	1 (2.6%)
Ethnicity		
American Indian or Alaskan	1 (1.1%)	0 (0.0%)
Asian	58 (63.7%)	23 (60.5%)
Black or African American	0 (0.0%)	0 (0.0%)
Hawaiian or other Pacific Islander	0 (0.0%)	0 (0.0%)
White	24 (26.4%)	12 (34.2%)
Other or multi-racial	8 (8.8%)	2 (5.3%)
Location		
North & South America	46 (50.5%)	28 (73.7%)
Asia	38 (41.8%)	6 (15.8%)
Europe	7 (7.7%)	4 (10.5%)

Fig. 5. Demographics and professional expertise and experience of the study participants.

3.2 Data Collection Procedure

Data collection started in August 2017 and ended in August 2018. After initial contact with interested developers, we used an online screening questionnaire to determine study eligibility and compensation (e.g., sufficient knowledge of Python, fluency in the English language, age over 18 years). Exclusion criteria included previous participation in a similar study using Java programming language [71] or previous participation in the current study, no knowledge of Python, under 18 years of age, lack of proficiency in English, and unwillingness to install and use the latest version of Mozilla Firefox, as our survey was tested on and compatible with this browser.

Eligible developers received a digital informed consent form, which disclosed study procedures, the minimal risk from study participation, and data privacy and anonymity. Each developer was assigned a unique, anonymized identifier to assure confidentiality. After providing their digital signature, developers received an audio test link. This test was implemented to ensure that proper data quality was captured for the audio recording during the cognitive testing. It was instituted

following the failure to collect audio data from a large fraction of the participants in the earlier Java study [71]. Participants were instructed to read three sentences into their microphone. The resulting test audio was reviewed by research staff, and if the audio quality was good, participants received a personalized link to the online assessment. If the audio was not captured or of poor quality, then research staff worked with the participant to address audio issues. This revision of the data collection methodology was successful in addressing the shortcomings of the earlier study [71].

After completing the puzzles, the participants were sent an additional survey on software security. This survey aimed to gather information regarding the participants' respective organizations' focus, organization size, organization software products developed, and also asked security-related questions.

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 and cognitive assessment). All data collection took place in a location of the participants' choosing. To increase ecological validity (the likelihood that the study's results can generalize to the real world), participants were informed they could use outside resources as assistance and were asked to report the resources used throughout the survey. The participants were not explicitly given access to an environment to execute the code, though as they had access to outside resources, they could have elected to do this. No participant reported doing this. The participants were not informed that the study was about code security.

The study procedure comprised five parts. The first part, programming puzzles (recall Section 2) involved responding to the programming puzzles and related questions. Participants were told that the puzzles were designed to examine how developers interpret and reason about code; they were not informed about the nature or presence of blindspots in the code. Identical to our prior, Java-based study [71], the second part of the study assessed participant demographics; the third part professional experience and expertise; the fourth part was a personality assessment that utilized the Big Five Inventory (BFI) questionnaire [52]; and the final part consisted of a cognitive assessment, comprising the Oral Symbol Digit Test from the NIH Toolbox [44] and the Brief Test of Adult Cognition by Telephone (BTACT) [98]. The average time to complete the entire study ranged between 30 to 90 minutes.

4 EXPERIMENTAL DESIGN AND RESULTS

Our analysis aims to answer six research questions:

RQ1: Are developers less likely to correctly solve puzzles with API functions containing blindspots than puzzles with innocuous functions? Does the underlying programming language have an effect?

RQ2: Do developers perceive puzzles with API functions containing blindspots as more difficult, as less clear, and as less familiar than non-blindspot puzzles? Does the underlying programming language have an effect?

RQ3: Are developers less confident about their puzzle solution when working on puzzles with API functions containing blindspots than non-blindspot puzzles? Does the underlying programming language have an effect?

RQ4: Are developers with higher cognitive functions (e.g., reasoning, working memory, and processing speed) better at solving puzzles with API functions containing blindspots? Does the underlying programming language have an effect?

RQ5: Are developers with more professional experience and expertise better at solving puzzles with API functions containing blindspots? Does the underlying programming language have an effect?
RQ6: Are developers with higher levels of conscientiousness and openness and lower levels of neuroticism, extraversion, and agreeableness better at solving puzzles with API functions containing blindspots? Does the underlying programming language have an effect?

4.1 Analysis Methodology

In answering RQ1, RQ2, and RQ3, we used multilevel modeling. In answering RQ4, RQ5, and RQ6, we used ordinal logistic regression. We conducted all analyses using Stata 15.0 and applied the Wald test to determine statistical significance of main and interaction effects.

Unlike our prior, Java-based study [71], for all non-significant effects, we further conducted Bayesian statistic analyses to determine whether a given non-significant effect was more likely due to insensitivity of the data or reflected a preference to accept the null hypothesis [36]. In other words, for situations in which the p-value of the Wald test was too high for us to reject the null hypothesis, this analysis allowed us to estimate the likelihood that the p-value is high because of insufficient data versus because there is no underlying difference between the distributions being compared. A non-significant effect with a Bayes Factor between 0.33 and 1 indicates data insensitivity (that is, our data is insufficient to draw a conclusion); a non-significant effect with a Bayes factor lower than 0.33 indicates preference to accept the null hypothesis (that is, that the two distributions being compared actually come from the same underlying distribution). As a hypothetical example, consider the situation in which we have two groups of developers participating in our study: one wearing red hats and one wearing green hats. The Wald test determines whether the difference in the number of puzzles the two groups solve correctly is statistically significant—that is, the p-value is the probability that the two distributions actually come from the same underlying distribution of developers. If the p-value is sufficiently low, then we can conclude that there exists a statistically significant difference between red-hat and green-hat developers, with respect to the number of puzzles solved correctly. But if the p-value is too high, then the Bayes statistic analysis allows us to determine the likelihood that the two distributions of developers are actually coming from the same underlying distribution, versus the possibility that our data is simply insufficient to tell if the distributions are, in fact, different. This extension of the methodology used in the original Java-based study we are replicating allows for a more thorough analysis of our results.

We now answer each of the six research questions. For each question, we present the data and analysis for the Python puzzles, recap and compare to the findings with respect to the Java puzzles [71], and analyze the languages' effect. For the direct comparison between the two programming languages, we applied the same analytic approach as just described, but added programming language (Python versus Java) as a moderator in the models. For these analyses, we were particularly interested in the interaction of programming language in each model.

4.2 Do Blindspots Make Programming Tasks More Difficult?

To answer whether developers are less likely to correctly solve puzzles with API functions containing blindspots than puzzles without blindspots (RQ1), we used multilevel logistic regression. The underlying data were hierarchical: each set of six puzzles (level 1 data), nested within each developer (level 2 data). The independent variable was the presence of a blindspot (0 for no blindspot, 1 for blindspot), and the dependent variable was whether the developer solved the puzzle (0 for

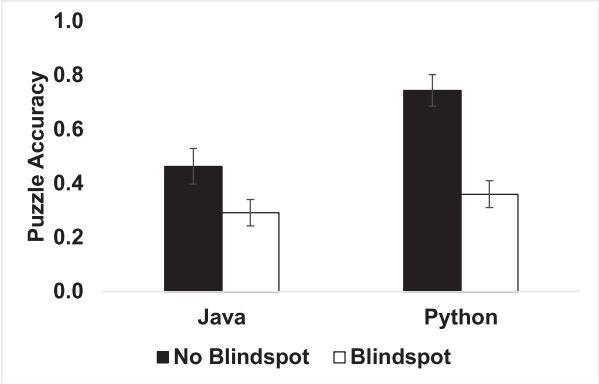

Fig. 6. Interaction effect of the programming language and the presence of a blindspot on puzzle accuracy. The x-axis shows the two programming languages, Java and Python. The y-axis shows the puzzle accuracy (the probability of the participant solving the puzzle correctly). Error bars represent 95% confidence intervals.

incorrect, 1 for correct). In this model, we also considered the random effect of the intercept to accommodate for inter-individual differences in overall puzzle accuracy. That is, our model accounts for the variability in each participant's overall puzzle solving accuracy.

For Python, the 129 participants correctly solved 74% of the puzzles without blindspots, but only 36% of the puzzles with blindspots. The effect of the presence of blindspot was significant (Wald $\chi^2(1) = 80.61$, $p < 0.001$), rejecting the null hypothesis that the puzzle solving accuracy for puzzles with blindspots and without came from the same distribution. Thus, participants were more than twice as likely to solve a Python puzzle correctly if that puzzle used an API without a blindspot than if it used one with a blindspot.

These results are similar to the observations for Java API puzzles [71]: The 109 participants correctly solved 46% of the puzzles without blindspots, but only 29% of the puzzles with blindspots, which was similarly a significant difference (Wald $\chi^2(1) = 20.60$, $p < 0.001$). Thus, participants were about 1.6 times as likely to solve a Java puzzle correctly if that puzzle used an API without a blindspot than if it used one with a blindspot.

We combined the Java and Python datasets and treated the programming language (Python versus Java) as a moderator in the models. Expectedly, we again found a statistically significant difference in puzzle accuracy for puzzles with and without APIs with blindspots (Wald $\chi^2(1) = 94.84$, $p < 0.001$). The interaction between programming language and presence of blindspot (Wald $\chi^2(1) = 13.93$, $p < 0.001$) was significant. Figure 6 depicts this interaction and shows that, overall, developers were less likely to accurately solve puzzles with blindspots than ones without blindspots, with this effect more pronounced for Python ($B = -1.78$ (the slope of the line between the predictor variable and the dependent variable), $z = -8.98$ (z-score describes the deviation from the mean in number of standard deviations), $p < 0.001$, odds ratio = 0.17) than for Java ($B = -0.81$, $z = -4.54$, $p < 0.001$, odds ratio = 0.44).

(RQ1) Overall, our statistical analysis supports that developers were significantly more successful in correctly solving puzzles that used APIs without blindspots than with blindspots, for both programming languages, with this effect more pronounced for Python than Java.

We next looked at the three kinds of APIs used in our puzzles (I/O, cryptography, and string manipulation) and the cyclomatic complexity of the puzzles and how these variables affected the

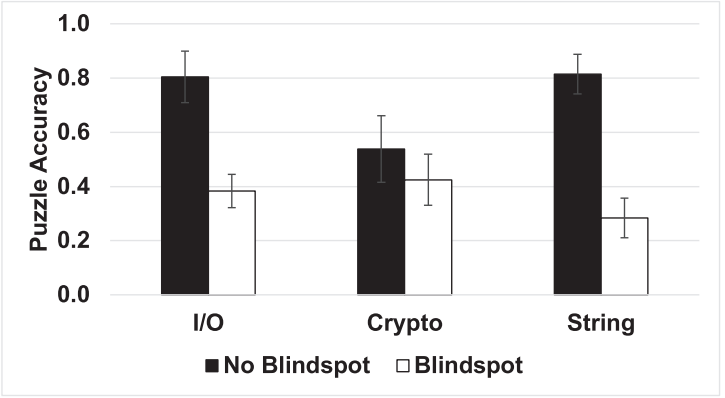

Fig. 7. Interaction effect of the API type and the presence of a blindspot on puzzle accuracy. The x-axis shows the three types of API usage: I/O, crypto, and string manipulation. The y-axis shows the puzzle accuracy (the probability of the participant solving the puzzle correctly). Error bars represent 95% confidence intervals after Bonferroni correction of the p-value.

developers' ability to solve puzzles. To control for family-wise type-I error inflation due to testing of multiple dependent models (i.e., models that share the same dependent variable), we applied Bonferroni correction for the p-value threshold to determine statistical significance ($p < 0.025$).

For Python, adding the categorical variable API usage type, we measured its interaction with the presence of an API with a blindspot in a puzzle as a predictor for whether the developer correctly solved the puzzle. The main effect of API usage type was not significant (Wald $\chi^2(2) = 6.55, p = 0.04$). However, its interaction with the presence of a blindspot was significant (Wald $\chi^2(2) = 21.06, p < 0.001$). Figure 7 shows that while developers were always less likely to accurately solve puzzles with APIs containing blindspots than puzzles without blindspots, this difference in accuracy was more pronounced for puzzles using API functions that involved I/O and string manipulation than those that involved crypto.

In the Java-puzzle study [71], the main effect of the presence of blindspot was not significant (Wald $\chi^2(2) = 0.91, p = 0.34$), but the main effect of API usage type (Wald $\chi^2(2) = 10.64, p = 0.005$) and its interaction with the presence of blindspot (Wald $\chi^2(2) = 24.81, p < 0.001$) were significant. Accuracy was higher for puzzles without blindspots than for ones with blindspots with an API function that involved I/O. Accuracy was comparable in both puzzles with and without blindspots with API functions that involved Crypto, String manipulation.

We then, again, treated the language as a moderator to explore the extent to which API usage type moderated accuracy for puzzles with blindspots versus ones without blindspots for Python and Java. Again, we applied Bonferroni correction for the p-value threshold to determine statistical significance ($p < 0.025$). The three-way interaction between presence of blindspot, API usage type, and programming language was significant (Wald $\chi^2(2) = 12.38, p = 0.002$). Figure 8 shows that, while in Python, developers were less likely to correctly solve puzzles with blindspots than without blindspots for all three API usage types; in Java, this accuracy difference only held in puzzles with I/O API functions, but not puzzles with Crypto and String API functions.

(RQ1, for different API types) Our statistical analysis supports that for Python, developers were overall more accurate solving puzzles without blindspots than puzzles with blindspots for all three API usage types, but for Java, that was only the case for puzzles with I/O API functions, not puzzles with Crypto and String API functions.

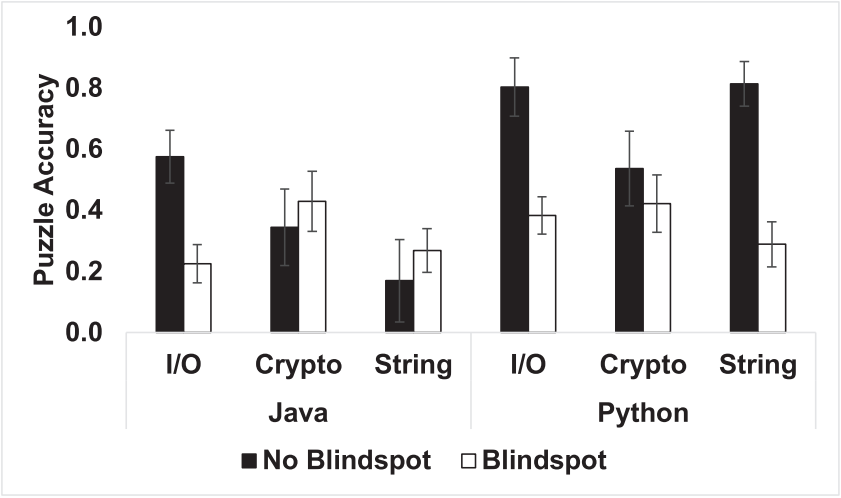

Fig. 8. Interaction effect of the programming language, API usage type, and the presence of a blindspot on puzzle accuracy. The x-axis shows the three types of API usage: I/O, Crypto, and String, and differentiates between Java (left) and Python (right) puzzles. The y-axis shows the puzzle accuracy (the probability of the participant solving the puzzle correctly). Error bars represent 95% confidence intervals after Bonferroni correction of the p-value.

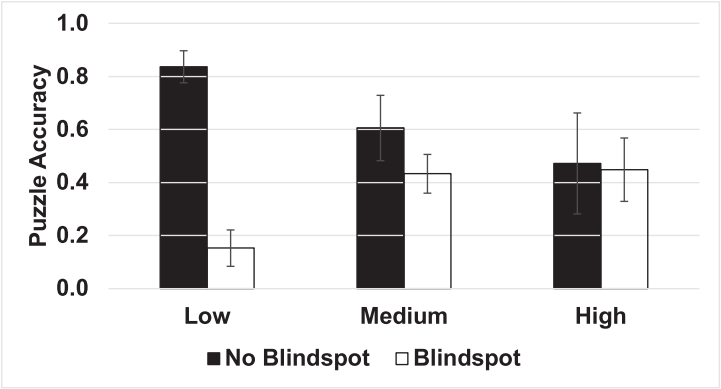

Fig. 9. Interaction effect of cyclomatic complexity and presence of a blindspot on puzzle accuracy. The x-axis shows the puzzle's level of cyclomatic complexity. The y-axis shows the puzzle accuracy (the probability of the participant solving the puzzle correctly). Error bars represent 95% confidence intervals after Bonferroni correction of the p-value.

Next, adding the categorical variable cyclomatic complexity (1 for low, 2 for medium, 3 for high), we measured its interaction with the presence of an API with a blindspot in a puzzle as a predictor for whether the developer correctly solved the puzzle. Again, while the main effect of cyclomatic complexity was not significant (Wald $\chi^2(2) = 0.97, p = 0.62$), its interaction with the presence of a blindspot was significant (Wald $\chi^2(2) = 37.34, p < 0.001$). Figure 9 shows that for puzzles of low cyclomatic complexity, the difference in accuracy was very large, whereas for puzzles of medium complexity, the difference in accuracy was small. For puzzles with high cyclomatic complexity, there was no measured difference. We speculate that a possible explanation for this behavior is that developers may be more careless with seemingly simpler tasks, as we describe in Section 5.

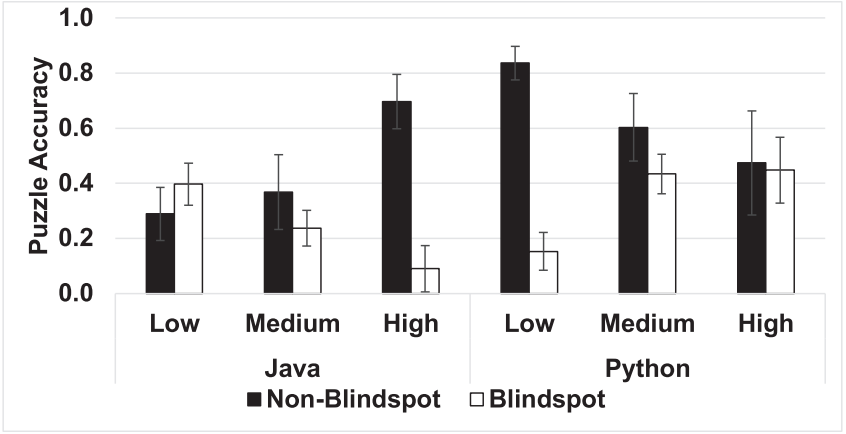

Fig. 10. Interaction effect of the programming language, cyclomatic complexity, and presence of a blindspot on puzzle accuracy. The x-axis shows the puzzle's level of cyclomatic complexity and differentiates between Java (left) and Python (right) programmers. The y-axis shows the puzzle accuracy (the probability of the participant solving the puzzle correctly). Error bars represent 95% confidence intervals after Bonferroni correction of the p value.

In the Java-puzzle study [71], the main effect of cyclomatic complexity was not significant (Wald $\chi^2(2) = 0.74$, $p = 0.69$), but the main effect of the presence of a blindspot (Wald $\chi^2(2) = 23.95$, $p < 0.001$) and its interaction with cyclomatic complexity (Wald $\chi^2(2) = 30.1$, $p < 0.001$) were significant. Accuracy was higher for puzzles without blindspots than those with blindspots 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 APIs with blindspots, the less likely developers were to correctly solve the puzzle.

Again, we then explored the extent to which cyclomatic complexity moderated accuracy for puzzles with blindspots versus ones without blindspots for Python and Java. Again, we applied Bonferroni correction for the p-value threshold to determine statistical significance ($p < 0.025$). The three-way interaction between presence of blindspot, cyclomatic complexity, and programming language was significant (Wald $\chi^2(2) = 64.40$, $p < 0.001$). Figure 10 shows that, while in Java, the difference in accuracy for puzzles with and without blindspots increased with the cyclomatic complexity, in Python, this accuracy difference decreased with cyclomatic complexity. This significant difference in effect of cyclomatic complexity on the developers accuracy is surprising. Possible explanations include differences in the underlying languages, e.g., Python may instill additional false confidence in the developers, whereas Java's verbosity has the effect of discouraging developers from making rash decisions. This hypothesis is supported by our finding that developers rated Java puzzles as more complex than Python puzzles (see Section 4.3). Overall, these results suggest further study is warranted to understand the different ways in which code of differing cyclomatic complexity may affect programs in different languages and whether such differences can be used to reduce the pitfalls of developers creating security vulnerabilities as a result of API blindspots.

(RQ1, based on cyclomatic complexity) For Python, developers were more likely to correctly solve puzzles without blindspots than puzzles with blindspots for low-complexity (but not medium- and high-complexity) puzzles. For Java, the opposite was true, with developers more likely to correctly solve puzzles without blindspots than puzzles with blindspots for high-complexity (but not medium- or low-complexity) puzzles.

4.3 Blindspots' Effect on Difficulty, Clarity, Familiarity, and Confidence

Next, we were interested in learning if the way developers perceive the puzzles correlates with the developers' ability to solve the puzzles correctly. For RQ2, we considered the developers' self-reported perception of the puzzles as more difficult, less clear, and less familiar. For RQ3, we considered the developers' self-reported confidence in their solutions. Self-reporting is a common way to measure participants' perception, including confidence when seeking to understand the relationship between confidence and accuracy in performing tasks [51].

We used a set of multilevel regression models to accommodate for the hierarchical data structure (recall that each set of six puzzles (level 1 data) is nested within each developer (level 2 data)). We considered four models, each with a different dependent variable: numerical ratings of difficulty, clarity, familiarity, and confidence. In each model, we considered the random effect of the intercept to accommodate for inter-individual differences in overall ratings of the respective dimension.

For Python, we found no significant effects of presence of a blindspot in a puzzle on the four dependent variables (all $p > 0.05$), suggesting that developers' perception of the puzzles did not differ as a function of presence of a blindspot in the puzzle. The Bayesian statistical analyses showed that all non-significant effects supported the acceptance of the null hypothesis (all Bayes factors < 0.0007).

In the Java-puzzle study [71], the developers' perceptions did not differ as a function of the presence of blindspot in puzzles (all $p > 0.05$).

We then, again, combined the Java and Python datasets and treated the language as a moderator in the models. We found a significant effect of programming language on difficulty (Wald $\chi^2(1) = 22.11$, $p < 0.001$). That is, developers rated Java puzzles overall as more difficult than Python puzzles, providing some support for our earlier hypothesis that some Python puzzles may seem so simple to the developers that they are even more likely to overlook blindspots in APIs than they are in more complex puzzles. All other main and interaction effects in these models (one for each of difficulty, clarity, familiarity, and confidence) were not significant (all $p > 0.10$). The Bayesian statistical analyses showed that the Bayes Factor of the non-significant effect for confidence was 0.94, suggesting insensitivity of the present data to detect this effect, meaning that we cannot draw a conclusion for these factors. In contrast, the Bayes factors of the non-significant effects for clarity and familiarity supported the acceptance of the null hypothesis (all Bayes factors < 0.01).

> (RQ2 and RQ3) We find that whether the developers considered puzzles to be more difficult, less clear, and less familiar did not have an effect on their ability to correctly solve the puzzles. Our statistical analysis supports that developers rated Java puzzles as more difficult than Python puzzles. We find that puzzle clarity and familiarity had no effect on the developers' ability to solve the puzzles, whereas our data was insufficient to draw a conclusion with respect to the developers' confidence in their answer.

4.4 Developers' Traits Effect on Ability to Solve Puzzles

We next set out to determine if developers' traits affected their ability to correctly solve puzzles that use APIs with blindspots. For RQ4, we considered the developers' higher cognitive functions, such as reasoning, working memory, and processing speed. For RQ5, we considered the developers' professional experience and expertise. For RQ6, we considered the developers' levels of conscientiousness and openness and levels of neuroticism and agreeableness.

For this analysis, we constructed an ordinal outcome variable with a range of 0 to 4 consisting of the number of puzzles with blindpots that a developer solved correctly. (Recall that each developer attempted to solve 6 puzzles, 4 of which used APIs with blindspots.) We conducted ordinal logistic

regressions using this ordinal outcome variable to test the effect of each trait a developer possesses on their ability to solve puzzles with blindspots.

We conducted a separate model for each of the three research questions. For RQ4, measuring cognitive functions, the independent variables consisted of the Oral Symbol Digit Test from the NIH toolbox and the backward counting, backward digit span, category fluency, immediate word list recall, delayed word list recall, and number series from the Brief Test of Adult Cognition by Telephone (BTACT) [98]. Immediate word list recall is a measure of short-term memory, while delayed word recall is a measure of one's long-term memory. The Oral Symbol Digit Test is a measure of one's processing speed. The backward counting is a measure of immediate and delayed episodic memory, the backward digit span is a measure of working memory span, and the category fluency test measures verbal fluency, executive functioning, and speed of processing [60]. We measured inductive reasoning using a number series completion task [81, 85].

Among all cognitive measures, only the delayed word list recall task showed a significant effect ($B = 0.20$, $z = 2.37$, $p = 0.02$, odds ratio = 1.23). Developers with better long-term memory were more likely to correctly solve puzzles with blindspots (see Figure 11). None of the other predictors showed a significant effect (for each, $p > 0.10$). The Bayesian statistical analyses showed that all non-significant effects supported the acceptance of the null hypothesis (all Bayes factors < 0.002), meaning that the data support the conclusion that these factors do not affect developers' ability to correctly solve puzzles.

In the Java-puzzle study [71], the dataset was limited for cognitive measures due to missing data, and there were no significant effects observed for any of the three cognitive measures on blindspot puzzle accuracy (all $p > 0.05$).

When comparing the two programming languages directly regarding the impact of cognitive measures on the programmers' ability to solving puzzles, we examined each cognitive measure individually (i.e., in a separate model). In the prior study [70], a technical error led 35 Java developers to have missing data on some of the cognitive measures (an error our study avoids). For this analysis, none of the cognitive measures showed a significant interaction with programming language (all $p > 0.10$), suggesting that the null effect of cognitive abilities on solving blindspot puzzles did not vary as a function of programming language. The Bayesian statistical analyses showed that all non-significant interactions supported the acceptance of null hypothesis (all Bayes factors $< 5 \times 10^{-6}$), meaning that findings are consistent across Java and Python, and the programming language of the puzzles is unlikely to have an effect on which cognitive functions affect developers' ability to solve puzzles.

> (RQ4) We conclude that developers with better long-term memory recall are more likely to correctly solve puzzles with blindspots. Short-term memory, processing speed, episodic memory, and memory span do not affect developer's ability to correctly solve puzzles with blindspots. The programming language of the puzzles did not affect which cognitive measures have an effect on the developers ability to solve puzzles correctly.

For RQ5, measuring professional experience and expertise, the independent variables consisted of years of programming, technical proficiency score, and Python and Java skills. Four developers had missing data on (some of) the technical expertise or experience measures, so we excluded them from our analysis in this model, so this model was based on 125 samples. The Java analysis was based on the full 109 developers' samples [71].

For Python, none of the three predictors of technical expertise and experience predicted blindspot puzzle accuracy (for each, $p > 0.10$). The Bayesian statistical analyses showed that

all non-significant interactions supported the acceptance of the null hypothesis (all Bayes factors < 0.0001).

In the Java-puzzle study [71], none of the three predictors of experience and expertise predicted blindspot puzzle accuracy (all $p > 0.05$).

When directly comparing the two programming languages regarding the impact of technical expertise and experience on the programmers' ability to solve puzzles, none of the technical expertise and experience showed a significant interaction with programming language (all $p \geq 0.10$), suggesting that the null effect of technical expertise and experience on solving blindspot puzzles did not vary as a function of programming language. The Bayesian statistical analyses showed that all these non-significant effects supported the acceptance of the null hypothesis (all Bayes factors < 0.002), suggesting that there is no relationship between professional experience and expertise and the developers' ability to correctly solve puzzles.

(RQ5) We conclude that, surprisingly, professional experience and expertise do not improve the developers' ability to solve puzzles with blindspots correctly. This result was consistent across the two languages.

For RQ6, the independent variables consisted of five personality traits: agreeableness, conscientiousness, extraversion, neuroticism, and openness. We measured these personality traits use the Big Five Inventory (BFI) questionnaire [52]. The questionnaire is composed of 44 personality statements used to assess the five personality traits by having the participant rate the level they feel they endorse the given personality statement using a Likert scale. For this research question (unlike RQ5), our dataset covered all 129 participants.

For Python, none of the personality traites showed a significant effect (all $p > 0.18$). The Bayesian statistical analyses showed that all these non-significant effects supported the acceptance of the null hypothesis (all Bayes factors $< 1 \times 10^{-10}$), suggesting that there is no relationship between the personality traits and the developers' ability to correctly solve puzzles.

In the Java-puzzle study [71], 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 $p > 0.05$).

When directly comparing the two programming languages regarding the impact of personality traits on the programmers' ability to solve blindspot puzzles, the interaction between extraversion and programming language (Wald $\chi^2(1) = 5.95$, $p = 0.01$) as well as the interaction between openness and programming language (Wald $\chi^2(1) = 13.50$, $p < 0.001$) were significant. That is, developers with lower extraversion (see Figure 12) and higher openness (see Figure 13) were more accurate in solving Java blindspot puzzles, while these effects did not hold for Python puzzles. The interactions between agreeableness, conscientiousness, as well as neuroticism and programming language, however, were not significant (all $p > 0.09$). The Bayesian statistical analyses showed that all these non-significant interactions supported the acceptance of the null hypothesis (all Bayes factors $< 10^{-5}$).

(RQ6) We conclude that developers with lower extraversion and higher openness as personality traits were more accurate in solving Java puzzles with blindspots. In contrast, developers' personality traits were not associated with their accuracy to solve Python puzzles with blindspots.

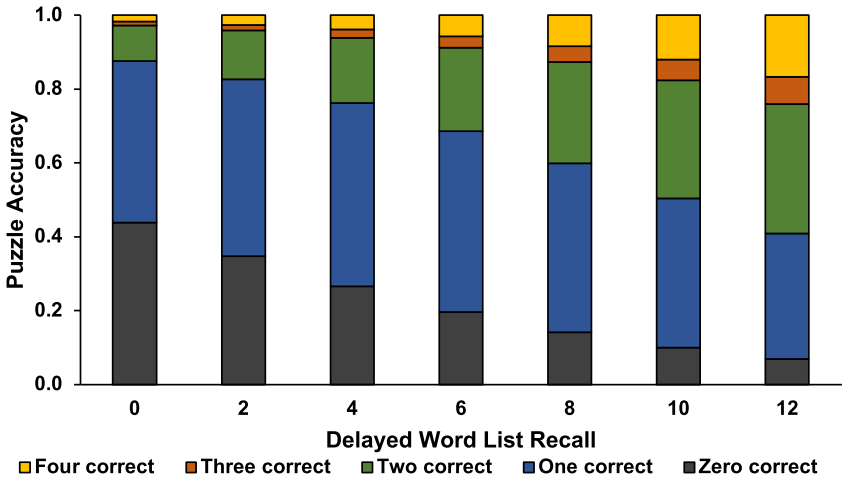

Fig. 11. Effect of long-term memory on puzzle solution accuracy for Python puzzles with blindspots. The x-axis represents the delayed word list recall task score (a measure of long-term memory), with higher scores reflecting better long-term memory. The y-axis shows the probability of correctly solving a given number (from 0 to 4) of blindspot puzzles; higher accuracy in solving blindspot puzzles is associated with darker colors.

5 DISCUSSION

Next, Section 5.1 discusses our findings and their implications, and Section 5.2 details the threats to the validity of our study.

5.1 Discussion of Findings and Implications

Most importantly, our study confirms that developers are less likely to correctly understand code that uses APIs with blindspots, even those that cause well-known vulnerabilities. This effect was most pronounced for APIs that dealt with I/O for Java; for Python, the effect was seen for all three API types. While our study's design does not determine *why* API blindspots make it more difficult for developers to understand code, the most likely reasons are either that the developers miss the potential vulnerability caused by the blindspot and assume the API will behave differently than it really does, or the developers are aware of the blindspot but the added complexity of the potentially unexpected behavior makes understanding it more difficult. Future research could work toward isolating the reasons blindspots cause developers to make mistakes.

Importantly, developers' expertise and experience did not help mitigate the risks associated with blindspots. Nor did the developers' perception of puzzle difficulty, clarity, and their familiarity with the involved concepts affect their success. Unfortunately, our data were insufficient for us to determine whether developers' confidence in their solution indicated a higher chance that their solution was correct. Given the large size of our user study, the fact that this result was inconclusive suggests that the effect of confidence may, at least, not be as strong as one might expect. One possible explanation for these observations is that, perhaps, expertise, experience, and perceptions about the puzzles may give developers false confidence in their solutions. For example, inexperienced developers perhaps make mistakes because of their inexperience; but experienced developers perhaps make similar mistakes because they are overconfident due to their experience and spend less care on examining the code. Similarly, developers who, for example, are unfamiliar with the involved concepts or who think the concepts are difficult may expand more care thinking about the code than those who are very familiar with the concepts or who find them easy.

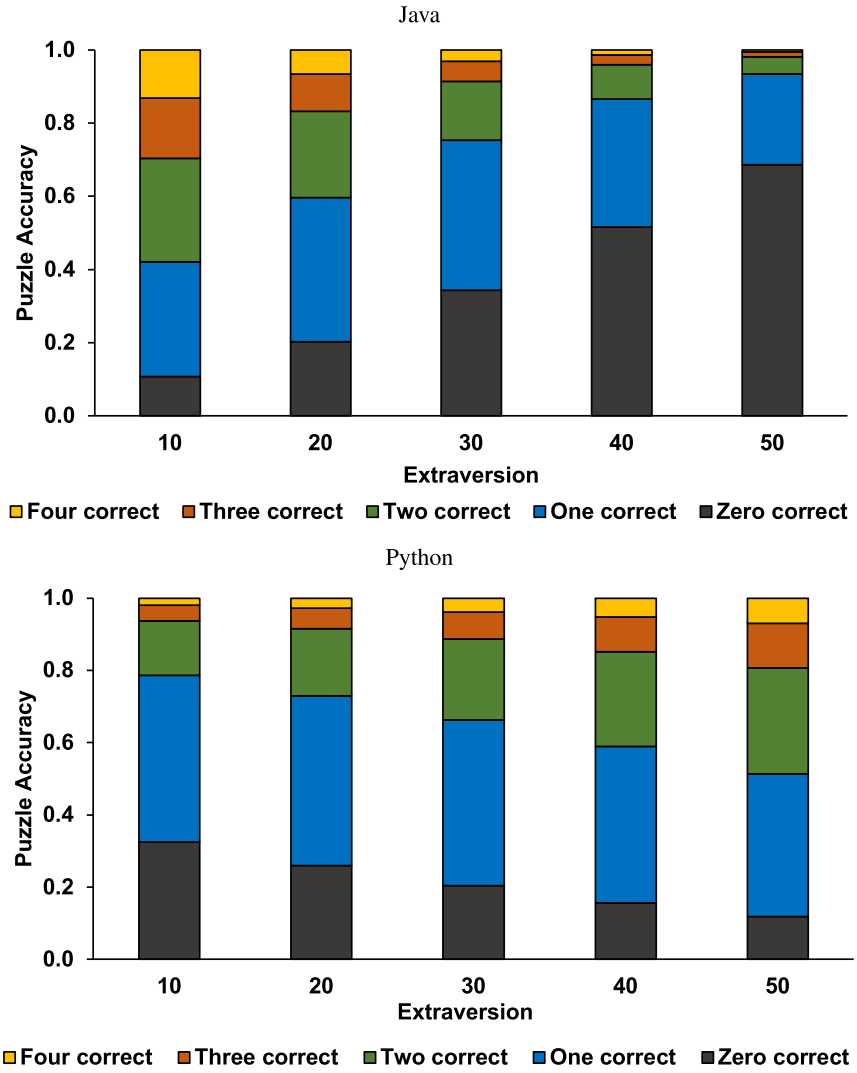

Fig. 12. Association between extraversion and accuracy for Java (top) and Python (bottom) puzzles with blindspots. The x-axis represents the level of extraversion, with higher scores reflecting more extraversion. The y-axis shows the probability of correctly solving a given number (from 0 to 4) of blindspot puzzles; higher accuracy in solving blindspot puzzles is associated with darker colors.

The only factor about the developers that had a consistent effect on their ability to correctly solve puzzles was their long-term memory. Developers with better long-term memory were more likely to correctly solve puzzles. We speculate that perhaps long-term memory allows developers to recall past experiences with the involved APIs with blindspots and apply that knowledge to more often solve the puzzles correctly.

Interestingly, we observed opposite results for Python and Java with respect to puzzle complexity. For Python, developers were far more often correct when solving low-complexity puzzles with APIs without blindspots than with blindspots, but there was virtually no difference for high-complexity. For Java, the opposite was true, with developers far more often correct when solving high-complexity puzzles with APIs without blindspots than with blindspots. We speculate that a

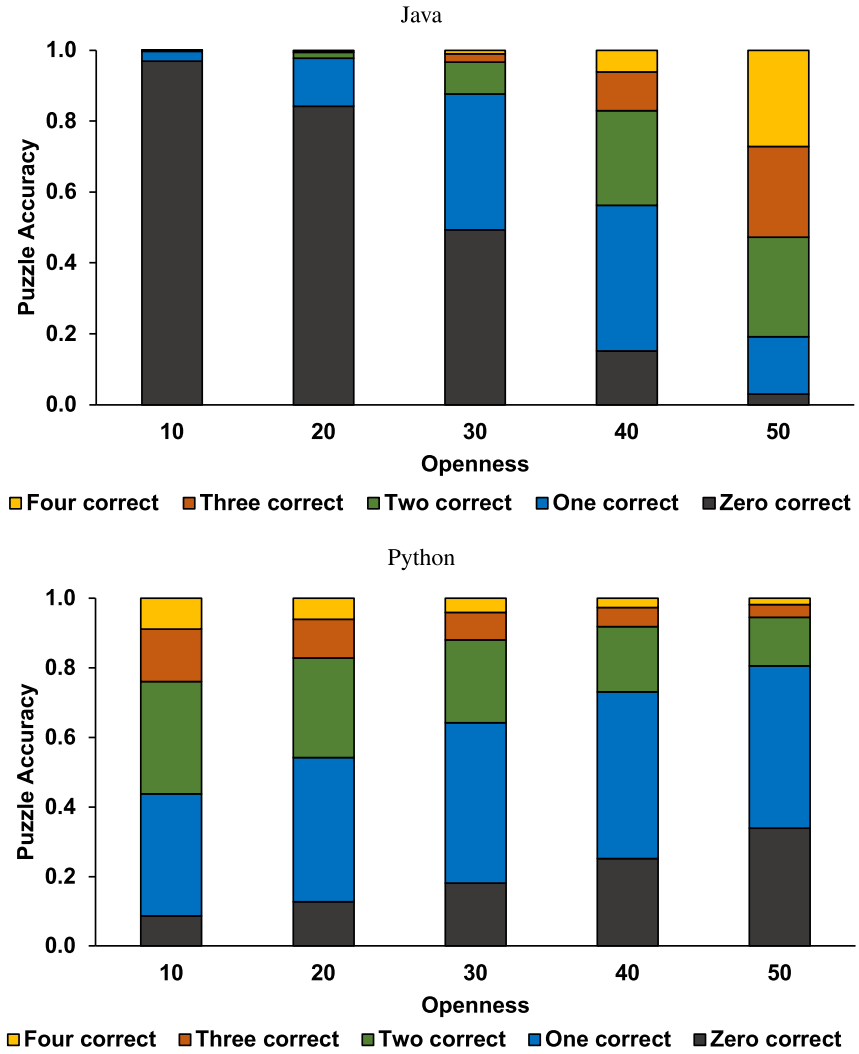

Fig. 13. Association between openness and accuracy for Java (top) and Python (bottom) puzzles with blindspots. The x-axis represents the level of openness, with higher scores reflecting more openness. The y-axis shows the probability of correctly solving a given number (from 0 to 4) of blindspot puzzles; higher accuracy in solving blindspot puzzles is associated with darker colors.

possible explanation for this behavior is that Python developers may be more careless with seemingly simpler tasks. Developers may perhaps have a false sense of security for low-complexity Python puzzles, which causes them more often to be blind to the blindspots. If our hypothesis is true, then it may suggest that the effects of at least some blindspots can be overcome by tools that simply remind developers to pay extra attention when dealing with potentially unsafe APIs, as opposed to requiring significant education or expertise with that particular API. The developers rated the Java puzzles to be more complex than the Python puzzles, so it is possible that the false sense of security does not surface for Java puzzles; instead, the more expected lower accuracy rate for more complex puzzles dominates. A related interesting observation is that for Java, the accuracy for puzzles without blindspots increased with complexity.

5.2 Threats to Validity

This article presents a replication study. Replication's central goal is to improve the external validity of research by replicating it in a different context. We focus here on replicating a study performed on puzzles involving Java APIs by developing puzzles for Python APIs and recruiting 192 developers as study subjects. To improve our study's internal validity, we reuse the methodology from the prior study [71] except in places where that prior study explicitly pointed out shortcomings that prevented desirable analyses, e.g., our study verifies that the user can properly record audio before engaging with the puzzles. To improve our study's ecological validity, we (as well as the Java study) use real-world APIs commonly reported in vulnerability databases [68, 87] or frequently discussed in developer fora [90]. The study's subjects were allowed to use outside resources, as they would in the real world.

The participants self-reported whether they were professionals or students. While it is possible that some students elected to lie to increase their payment, we believe this unlikely, as we asked the participants for their years of professional experience with Python, and if they lied, they may have feared being caught as inexperienced when solving the puzzles. It is possible to measure expertise more explicitly, e.g., via a test, but the study's length would either have become prohibitively long or we would have had to reduce the portion of the study in which developers solved the puzzles, and we elected to make the tradeoff to rely on self-reported expertise.

Our study's design considers APIs with and without blindspots, but it is possible that some APIs' blindspots are more severe than others. While we did not quantify the severity of a blindspot and measure the effects of that severity, future research could pursue this direction.

The results of our study draw conclusions about differences in effects of blindspots in Python and Java APIs. These differences can be caused by fundamental differences in the languages, differences in developers who use these languages, differences in relative puzzle difficulties across languages, and so on. As such, our claims do not differentiate between whether the language or the types of developers who use each language are the underlying reasons for our observations. It is fundamentally not possible to design a fully controlled experiment in the two languages that uses the same APIs with the same blindspots for both Python and Java, as the two languages use different APIs and different APIs have different blindspots. Out study varies the difficulty and API types across the puzzles across both languages and uses a large number of participants to mitigate this threat. A future study could consider more extensive ways of measuring API and puzzle complexity to account for some of the potentially confounding factors. The observed differences could even be accounted for by differences between the two specific sets of Python and Java developers who participated in our studies, but our use of a large number of participants (129 for Python and 109 for Java) mitigates this risk.

Python v3 uses somewhat different syntax from Python v2, and is not backward compatible, by design. Users who are proficient in one version are typically also proficient in the other. While our study did not explicitly check for the possibility that a participant is an expert with Python v3 but is inexperienced with v2, we believe this to be unlikely. All but one of our puzzles work with either version. Only Puzzle 1 from Figure 3 relies on the specific implementation of the `input` function in Python v2.7 and earlier, and the puzzle's syntax establishes the code as being written in Python v2.7.

6 RELATED WORK

Our study builds on our earlier work [71] that focused on how developers reason about Java APIs that have blindspots. This article develops 22 new puzzles that use Python APIs, 16 of which have known blindspots, and replicates our earlier study. This replication improves the data collection

methodology that allows for new analyses, allows us to generalize our findings across languages and explore differences between Python and Java, and uses Bayesian statistic analysis to make stronger claims than the prior study. The Java-based study has led to research into understanding why developers make security mistakes [75, 101], gaining insight into the developers' rationale in making API-use decisions [100] and evaluating the usability of security APIs [105]. Our replication study provides further support for that work.

Cappos et al. [28] proposed that software vulnerabilities are caused by blindspots in developers' heuristic-based decision-making mental models. Oliveira et al. [70] further showed that security is not a priority in the developers' mindsets while coding; however, that developers do 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. The rest of this section places our work in the context of API usability (Section 6.1), programming language design (Section 6.2), and the state of the practice of software development for security and privacy (Section 6.3).

6.1 API Usability

The study of API usability focuses on how to design APIs in a manner that reduces the likelihood of developer errors that can create software vulnerabilities [66]. Such research identifies common pitfalls in API design. For example, a study showed that the very popular factory design pattern [43] is detrimental to API usability, because when incorporated into an API it was difficult to use [38].

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.

Stylos and Clarke [93] 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"* [63, 86]), 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 [30]. Based on these findings they designed and implemented a Java language extension to realize these benefits [29]. However, to have a positive effect on usability and security, immutability needs to be carefully designed, with usability as a first-class requirement [104].

Recent work has investigated the usability of cryptographic APIs. Nadi et al. [67] 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 focuses on understanding blindspots that developers experience while working with general classes of API functions.

A significant body work has focused on automatically inferring models of API use or of APIs themselves to document API use practice. This work has spanned serial systems [10–12, 15, 34, 45, 59], distributed systems [13, 14], and resource-constrained systems [69]. Such work typically uses execution traces to capture API use patterns, which can also be used to find the location of a defect [35, 57] and the root cause of a defect [55]. Visualizing executions of models of groups of executions can further help locate bugs [57] and understand software behavior [16, 17]. In con-

trast, our work focuses on understanding how developers act when faced with APIs that contain blindspots and is complementary to these techniques that analyze APIs themselves or patterns of how developers use them.

6.2 Programming Language Design

Usability in programming language design has been a long-standing concern. Stefik and Siebert [92] showed that syntax used in a programming language was a significant barrier for novices. Research has also empirically compared programming languages, in particular, for whether some languages cause developers to create more bugs than others [9]. Some languages are designed to make it impossible to make certain kinds of errors, e.g., Java makes certain memory-use errors impossible by automatically managing memory use. Other languages, e.g., Coq [95] and HOL4 [89], are designed to enable formal verification, mathematically proving program correctness using proof assistants and theorem provers. While this is a highly manual and arduous process, recent advances in fully automating proof generation [39, 40, 82, 111] show promise. 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.

It is common for systems to exhibit emerging properties. Such properties are not explicitly coded by the developers into the system, but emerge either by design from the language, deployment frameworks, or APIs used by the system [19, 23–26], or without intention from interactions of the system's components. These emerging properties, particularly the latter kind, may surprise developers and could result in blindspots. We have addressed blindspots focusing on a single API, but future work should similarly explore blindspots caused by potential component interactions.

6.3 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). They found that developers tend to prioritize security tools over privacy policies, mostly because the language of privacy policies is so obscure [8].

Xie et al. [110] 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. [107] 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 [49] 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. [72] 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 fora (e.g., Stack Overflow) can lead to software vulnerabilities [3, 41, 99].

Certain aspects of software systems, such as security or fairness [27, 37, 42], are not only difficult for developers to reason about, it can be difficult for end-users to understand and describe the relevant requirements [50]. Recent work has aimed to develop APIs for components that can provide guarantees, e.g., that the component will not exhibit racist or sexist behavior when applied to future inputs [5, 46, 64, 96]. While early work has looked at whether such APIs help or hurt developers' and data scientists' work on improving system fairness [53, 54], fully understanding the implications of such APIs remains an open problem. Systems that automatically [6, 42] and manually [97] test systems for these properties are likely to help developers reason about their uses of these APIs.

Recent studies have investigated the need and type of interventions required for developers to adopt secure software development practices. Xie et al. [109] 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 [65, 106, 108]. Meanwhile the way that tools communicate with developers has been shown to be a critical aspect of successful adoption [56].

Research on the effects of external software security consultancy suggests [76] that a single time-limited involvement of developers with security awareness programs is generally ineffective in the long-term. Poller et al. [77] 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. [102] 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.

Most industrial and open-source development happens collaboratively, which can lead to other pitfalls, such as collaborative conflicts. Tools that improve developer-awareness can help avoid such pitfalls [20–22, 83, 84]. We envision similar tools can be built to warn developers of blindspots in the APIs they are using, helping them avoid introducing vulnerabilities into their code.

Our work on studying how developers use APIs and how blindspots affect them, and their code, complements these studies that have not focused on blindspots. Together, this work is building a better understanding of the developers' processes and how the tools and APIs at their disposal improve, or stand in their way, of writing high-quality, secure programs.

7 CONTRIBUTIONS

We have replicated our earlier, Java-based controlled experiment studying the effect of APIs with blindspots on developers [71]. Our replication applies to Python and involves 129 new developers and 22 new APIs. We found that APIs with blindspots statistically significantly reduce the developers' ability to correctly reason about the APIs in both languages, but that the effect is more pronounced for Python. Professional experience and expertise failed to mitigate this reduction, with long-term professionals with many years of experience making mistakes as often as relative novices. These findings suggest that blindspots in APIs are a serious problem across languages and that experience and education alone are insufficient to overcome it. Tools are needed to help developers recognize blindspots in APIs as they write code that uses those APIs, warning the developers and reducing the risk of the introduction of vulnerabilities.

Interestingly, for Java, the ability to correctly reason about APIs with blindspots improved with complexity of the code, whereas for Python, the opposite was true. We hypothesize that Python developers are less likely to notice potential for vulnerabilities in complex code than in simple code, whereas Java developers are more likely to recognize the extra complexity and apply more care, but are more careless with simple code. This finding suggests that, while blindspots likely have negative effects across programming languages, there are important differences among languages, warranting both further studies of more languages with respect to blindspots and deeper studies into whether the blindspots' unique effects on each language can be used to reduce the potentially resulting vulnerabilities.

ACKNOWLEDGMENTS

We wish to thank Daniela Oliveira, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong, and Justin Cappos for their contributions to data collection, puzzle creation, and the Java-based study.

REFERENCES

[1] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. 2017. Comparing the usability of cryptographic APIs. In *IEEE Symposium on Security and Privacy (SP'17)*. 154–171. DOI: https://doi.org/10.1109/SP.2017.52

[2] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. 2016. You get where you're looking for: The impact of information sources on code security. In *IEEE Symposium on Security and Privacy (SP'16)*. 289–305. DOI: https://doi.org/10.1109/SP.2016.25

[3] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. 2017. How internet resources might be helping you develop faster but less securely. *IEEE Secur. Priv.* 15, 2 (Mar. 2017), 50–60. DOI: https://doi.org/10.1109/MSP.2017.24

[4] Y. Acar, S. Fahl, and M. L. Mazurek. 2016. You are not your developer, either: A research agenda for usable security and privacy research beyond end users. In *IEEE Cybersecurity Development (SecDev'16)*. 3–8. DOI: https://doi.org/10.1109/SecDev.2016.013

[5] Alekh Agarwal, Alina Beygelzimer, Miroslav Dudík, John Langford, and Hanna Wallach. 2018. A reductions approach to fair classification. In *International Conference on Machine Learning (ICML'18)*, Vol. PMLR 80. 60–69.

[6] Rico Angell, Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2018. Themis: Automatically testing software for discrimination. In *Demonstrations Track at the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. 871–875. DOI: https://doi.org/10.1145/3236024.3264590

[7] John Anvik, Lyndon Hiew, and Gail C. Murphy. 2006. Who should fix this bug? In *International Conference on Software Engineering (ICSE'06)*. 361–370. DOI: https://doi.org/10.1145/1134285.1134336

[8] Rebecca Balebako, Abigail Marsh, Jialiu Lin, Jason I Hong, and Lorrie Faith Cranor. 2014. The privacy and security behaviors of smartphone app developers. In *Workshop on Usable Security*. Internet Society.

[9] Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. 2019. On the impact of programming languages on code quality: A reproduction study. *ACM Trans. Program. Lang. Syst.* 41, 4 (Oct. 2019). DOI: https://doi.org/10.1145/3340571

[10] Ivan Beschastnikh, Jenny Abrahamson, Yuriy Brun, and Michael D. Ernst. 2011. Synoptic: Studying logged behavior with inferred models. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering Tool Demonstration Track (ESEC/FSE'11)*. 448–451. DOI: https://doi.org/10.1145/2025113.2025188

[11] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. 2013. Unifying FSM-inference algorithms through declarative specification. In *International Conference on Software Engineering (ICSE'13)*. 252–261. DOI: https://doi.org/10.1109/ICSE.2013.6606571

[12] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. 2015. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Trans. Softw. Eng.* 41, 4 (Ap. 2015), 408–428. DOI: https://doi.org/10.1109/TSE.2014.2369047

[13] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In *International Conference on Software Engineering (ICSE'14)*. 468–479. DOI: https://doi.org/10.1145/2568225.2568246

[14] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, Arvind Krishnamurthy, and Thomas E. Anderson. 2011. Mining temporal invariants from partially ordered logs. *ACM SIGOPS Oper. Syst. Rev.* 45, 3 (Dec. 2011), 39–46. DOI: https://doi.org/10.1145/2094091.2094101

[15] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*. 267–277. DOI: https://doi.org/10.1145/2025113.2025151

[16] Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2020. Visualizing distributed system executions. *ACM Trans. Softw. Eng. Methodol.* 29, 2 (Mar. 2020), 9:1–9:38. DOI: https://doi.org/10.1145/3375633

[17] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2016. Debugging distributed systems. *Commun. ACM* 59, 8 (Aug. 2016), 32–37. DOI: https://doi.org/10.1145/2909480

[18] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. *Reversible Debugging Software*. Technical Report. University of Cambridge, Judge Business School.

[19] Yuriy Brun, George Edwards, Jae young Bang, and Nenad Medvidovic. 2011. Smart redundancy for distributed computation. In *International Conference on Distributed Computing Systems (ICDCS'11)*. 665–676. DOI: https://doi.org/10.1109/ICDCS.2011.25

[20] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Crystal: Precise and unobtrusive conflict warnings. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering Tool Demonstration Track (ESEC/FSE'11)*. 444–447. DOI: https://doi.org/10.1145/2025113.2025187

[21] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive detection of collaboration conflicts. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*. 168–178. DOI: https://doi.org/10.1145/2025113.2025139

[22] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2013. Early detection of collaboration conflicts and risks. *IEEE Trans. Softw. Eng.* 39, 10 (Oct. 2013), 1358–1375. DOI: https://doi.org/10.1109/TSE.2013.28

[23] Yuriy Brun and Nenad Medvidovic. 2007. An architectural style for solving computationally intensive problems on large networks. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS'07)*. DOI: https://doi.org/10.1109/SEAMS.2007.4

[24] Yuriy Brun and Nenad Medvidovic. 2007. Fault and adversary tolerance as an emergent property of distributed systems' software architectures. In *2nd International Workshop on Engineering Fault Tolerant Systems (EFTS'07)*. 38–43. DOI: https://doi.org/10.1145/1316550.1316557

[25] Yuriy Brun and Nenad Medvidovic. 2012. Keeping data private while computing in the cloud. In *5th International Conference on Cloud Computing (CLOUD'12)*. 285–294. DOI: https://doi.org/10.1109/CLOUD.2012.126

[26] Yuriy Brun and Nenad Medvidovic. 2013. Entrusting private computation and data to untrusted networks. *IEEE Trans. Depend. Secure Comput.* 10, 4 (July/Aug. 2013), 225–238. DOI: https://doi.org/10.1109/TDSC.2013.13

[27] Yuriy Brun and Alexandra Meliou. 2018. Software fairness. In *New Ideas and Emerging Results Track at the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. 754–759. DOI: https://doi.org/10.1145/3236024.3264838

[28] Justin Cappos, Yanyan Zhuang, Daniela Oliveira, Marissa Rosenthal, and Kuo-Chuan Yeh. 2014. Vulnerabilities as blind spots in developer's heuristic-based decision-making processes. In *New Security Paradigms Workshop (NSPW'14)*. 53–62. DOI: https://doi.org/10.1145/2683467.2683472

[29] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2017. Glacier: Transitive class immutability for Java. In *International Conference on Software Engineering (ICSE'17)*. 496–506. DOI: https://doi.org/10.1109/ICSE.2017.52

[30] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. 2016. Exploring language support for immutability. In *International Conference on Software Engineering (ICSE'16)*. 736–747. DOI: https://doi.org/10.1145/2884781.2884798

[31] Common Weakness Enumeration 2011. Common Weakness Enumeration (CWE)/SANS Top 25 Most Dangerous Software Errors. Retrieved from http://cwe.mitre.org/top25/.

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

[33] Barthélémy Dagenais and Martin P. Robillard. 2010. Creating and evolving developer documentation: Understanding the decisions of open source contributors. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*. 127–136. DOI: https://doi.org/10.1145/1882291.1882312

[34] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. 2010. Generating test cases for specification mining. In *International Symposium on Software Testing and Analysis (ISSTA'10)*. 85–96. DOI: https://doi.org/10.1145/1831708.1831719

[35] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight defect localization for Java. In *European Conference on Object Oriented Programming (ECOOP'05)*. 528–550. DOI: https://doi.org/10.1007/11531142_23

[36] Zoltan Dienes. 2014. Using Bayes to get the most out of non-significant results. *Front. Psychol.* 5 (2014), 781:1–781:17. DOI: https://doi.org/10.3389/fpsyg.2014.00781

[37] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. 2012. Fairness through awareness. In *Innovations in Theoretical Computer Science Conference (ITCS'12)*. 214–226.

[38] Brian Ellis, Jeffrey Stylos, and Brad Myers. 2007. The factory pattern in API design: A usability evaluation. In *International Conference on Software Engineering (ICSE'07)*. 302–312. DOI: https://doi.org/10.1109/ICSE.2007.85

[39] Emily First and Yuriy Brun. 2022. Diversity-driven Automated Formal Verification. In *44th International Conference on Software Engineering (ICSE'22)*. 749–761. DOI: https://doi.org/10.1145/3510003.3510138

[40] Emily First, Yuriy Brun, and Arjun Guha. 2020. TacTok: Semantics-aware proof synthesis. *Proc. ACM. Program. Lang. Object-Orient. Program. Syst. Lang. Applic.* 4 (Nov. 2020), 231:1–231:31. DOI: https://doi.org/10.1145/3428299

[41] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. 2017. Stack overflow considered harmful? The impact of copy paste on Android application security. In *IEEE Symposium on Security and Privacy (SP'17)*. 121–136. DOI: https://doi.org/10.1109/SP.2017.31

[42] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness testing: Testing software for discrimination. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'17)*. 498–510. DOI: https://doi.org/10.1145/3106237.3106277

[43] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc.

[44] Richard C. Gershon, Molly V. Wagster, Hugh C. Hendrie, Nathan A. Fox, Karon F. Cook, and Cindy J. Nowinski. 2013. NIH toolbox for assessment of neurological and behavioral function. *Neurology* 80, 11 Supplement 3 (2013), S2–S6.

[45] Carlo Ghezzi, Mauro Pezzè, Michele Sama, and Giordano Tamburrelli. 2014. Mining behavior models from user-intensive web applications. In *ACM/IEEE International Conference on Software Engineering (ICSE'14)*. 277–287. DOI: https://doi.org/10.1145/2568225.2568234

[46] Stephen Giguere, Blossom Metevier, Yuriy Brun, Bruno Castro da Silva, Philip S. Thomas, and Scott Niekum. 2022. Fairness guarantees under demographic shift. In *10th International Conference on Learning Representations (ICLR'22)*. Retrieved from https://openreview.net/forum?id=wbPObLm6ueA.

[47] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. 2017. Understanding misunderstandings in source code. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'17)*. 129–139. DOI: https://doi.org/10.1145/3106237.3106264

[48] GraphicsMagick 2017. GraphicsMagick 1.4 Heap-based Buffer Overflow Vulnerability. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2017-17915.

[49] M. Green and M. Smith. 2016. Developers are not the enemy!: The need for usable security APIs. *IEEE Secur. Priv.* 14, 5 (Sept. 2016), 40–46. DOI: https://doi.org/10.1109/MSP.2016.111

[50] Nina Grgic-Hlaca, Elissa M. Redmiles, Krishna P. Gummadi, and Adrian Weller. 2018. Human perceptions of fairness in algorithmic decision making: A case study of criminal risk prediction. In *World Wide Web Conference (WWW'18)*. 903–912. DOI: https://doi.org/10.1145/3178876.3186138

[51] Henry L. Roediger III and K. Andrew DeSoto. 2014. Confidence and memory: Assessing positive and negative correlations. *Memory* 22, 1 (2014), 76–91. DOI: https://doi.org/10.1080/09658211.2013.795974

[52] Oliver P. John and Sanjay Srivastava. 1999. The big five trait taxonomy: History, measurement, and theoretical perspectives. *Handb. Personal.: Theor. Res.* 2, 1999 (1999), 102–138.

[53] Brittany Johnson, Jesse Bartola, Rico Angell, Katherine Keith, Sam Witty, Stephen J. Giguere, and Yuriy Brun. 2020. Fairkit, fairkit, on the wall, who's the fairest of them all? Supporting data scientists in training fair models. *CoRR* abs/2012.09951 (2020).

[54] Brittany Johnson and Yuriy Brun. 2022. Fairkit-learn: A fairness evaluation and comparison toolkit. In *Demonstrations Track at the 44th International Conference on Software Engineering (ICSE'22)*. 70–74. DOI: https://doi.org/10.1145/3510454.3516830

[55] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2020. Causal Testing: Understanding Defects' Root Causes. In *International Conference on Software Engineering (ICSE'20)*. 87–99. DOI: https://doi.org/10.1145/3377811.3380377

[56] Brittany Johnson, Rahul Pandita, Justin Smith, Denae Ford, Sarah Elder, Emerson Murphy-Hill, Sarah Heckman, and Caitlin Sadowski. 2016. A cross-tool communication study on program analysis tool notifications. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. 73–84. DOI: https://doi.org/10.1145/2950290.2950304

[57] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *International Conference on Software Engineering (ICSE'02)*. 467–477. DOI: https://doi.org/10.1145/581339.581397

[58] Herb Krasner. 2020. The Cost of Poor Software Quality in the US: A 2020 Report. Retrieved from https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf.

[59] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic mining of specifications from invocation traces and method invariants. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'14)*. 178–189. DOI: https://doi.org/10.1145/2635868.2635890

[60] Margie E. Lachman, Stefan Agrigoroaei, Patricia A. Tun, and Suzanne L. Weaver. 2014. Monitoring cognitive functioning: Psychometric properties of the brief test of adult cognition by telephone. *Assessment* 21, 4 (2014). DOI: https://doi.org/10.1177/1073191113508807

[61] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable statistical bug isolation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. 15–26. DOI: https://doi.org/10.1145/1065010.1065014

[62] Thomas J. McCabe. 1976. A complexity measure. *IEEE Trans. Softw. Eng.* 2, 4 (July 1976), 308–320. DOI: https://doi.org/10.1109/TSE.1976.233837

[63] Joe McManus and Sandy Shrum. 2015. SEI CERT Oracle Coding Standard for Java. Retrieved from https://wiki.sei.cmu.edu/confluence/display/java/JavaCodingGuidelines.

[64] Blossom Metevier, Stephen Giguere, Sarah Brockman, Ari Kobren, Yuriy Brun, Emma Brunskill, and Philip Thomas. 2019. Offline contextual bandits with high probability fairness guarantees. In *Annual Conference on Neural Information Processing Systems (NeurIPS), Advances in Neural Information Processing Systems 32*. 14893–14904.

[65] Emerson Murphy-Hill, Da Young Lee, Gail C. Murphy, and Joanna McGrenere. 2015. How do users discover new tools in software development and beyond? *Comput. Supp. Coop. Work* 24, 5 (01 Oct. 2015), 389–422. DOI: https://doi.org/10.1007/s10606-015-9230-9

[66] Brad A. Myers and Jeffrey Stylos. 2016. Improving API usability. *Commun. ACM* 59, 6 (May 2016), 62–69. DOI: https://doi.org/10.1145/2896587

[67] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through hoops: Why do Java developers struggle with cryptography APIs? In *International Conference on Software Engineering (ICSE'16)*. 935–946. DOI: https://doi.org/10.1145/2884781.2884790

[68] National Vulnerability Database 1999. National Vulnerability Database. Retrieved from https://nvd.nist.gov/.

[69] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. 2014. Behavioral resource-aware model inference. In *IEEE/ACM International Conference on Automated Software Engineering (ASE'14)*. 19–30. DOI: https://doi.org/10.1145/2642937.2642988

[70] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. 2014. It's the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In *Annual Computer Security Applications Conference (ACSAC'14)*. 296–305. DOI: https://doi.org/10.1145/2664243.2664254

[71] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong, Justin Cappos, Yuriy Brun, and Natalie C. Ebner. 2018. API blindspots: Why experienced developers write vulnerable code. In *USENIX Symposium on Usable Privacy and Security (SOUPS'18)*. 315–328. Retrieved from https://www.usenix.org/system/files/conference/soups2018/soups2018-oliveira.pdf.

[72] Marten Oltrogge, Yasemin Acar, Sergej Dechand, Matthew Smith, and Sascha Fahl. 2015. To pin or not to pin helping app developers bullet proof their TLS connections. In *USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, 239–254.

[73] Open Web Application Security Project 2013. The Open Web Application Security Project (OWASP) Top 10 Most Critical Web Application Security Risks. Retrieved from https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf.

[74] H. Orman. 2003. The Morris worm: A fifteen-year perspective. *IEEE Secur. Priv.* 1, 5 (Sept. 2003), 35–43. DOI: https://doi.org/10.1109/MSECP.2003.1236233

[75] James Parker, Michael Hicks, Andrew Ruef, Michelle L. Mazurek, Dave Levin, Daniel Votipka, Piotr Mardziel, and Kelsey R. Fulton. 2020. Build it, break it, fix it: Contesting secure development. *ACM Trans. Priv. Secur.* 23, 2 (Apr. 2020), 10:1–10:36. DOI: https://doi.org/10.1145/3383773

[76] Andreas Poller, Laura Kocksch, Katharina Kinder-Kurlanda, and Felix Anand Epp. 2016. First-time security audits as a turning point?: Challenges for security practices in an industry software development team. In *SIGCHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI AE'16)*. 1288–1294. DOI: https://doi.org/10.1145/2851581.2892392

[77] Andreas Poller, Laura Kocksch, Sven Türpe, Felix Anand Epp, and Katharina Kinder-Kurlanda. 2017. Can security become a routine?: A study of organizational change in an agile software development group. In *ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW'17)*. 2489–2503. DOI: https://doi.org/10.1145/2998181.2998191

[78] Muhammad Sajidur Rahman. 2016. An Empirical Case Study on Stack Overflow to Explore Developers' Security Challenges. Masters Report. Retrieved from http://krex.k-state.edu/dspace/handle/2097/34563.
[79] M. P. Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE Softw.* 26, 6 (Nov. 2009), 27–34. DOI: https://doi.org/10.1109/MS.2009.193
[80] Martin P. Robillard and Robert Deline. 2011. A field study of API learning obstacles. *Empir. Softw. Eng.* 16, 6 (Dec. 2011), 703–732. DOI: https://doi.org/10.1007/s10664-010-9150-8
[81] Timothy A. Salthouse and Kenneth A. Prill. 1987. Inferences about age impairments in inferential reasoning. *Psychol. Aging* 2, 1 (1987). DOI: https://doi.org/10.1037/0882-7974.2.1.43
[82] Alex Sanchez-Stern, Emily First, Timothy Zhou, Zhanna Kaufman, Yuriy Brun, and Talia Ringer. 2022. Passport: Improving automated formal verification using identifiers. *CoRR* abs/2204.10370 (2022).
[83] Anita Sarma, Zahra Noroozi, and André van der Hoek. 2003. Palantír: Raising awareness among configuration management workspaces. In *International Conference on Software Engineering (ICSE'03)*. 444–454. DOI: https://doi.org/10.1109/ICSE.2003.1201222
[84] Anita Sarma, David F. Redmiles, and André van der Hoek. 2012. Palantír: Early detection of development conflicts arising from parallel code changes. *IEEE Trans. Softw. Eng.* 38, 4 (2012), 889–908. DOI: https://doi.org/10.1109/TSE.2011.64
[85] K. Warner Schaie. 1996. *Intellectual Development in Adulthood: The Seattle Longitudinal Study*. Cambridge University Press, New York, NY.
[86] Secure Coding Guidelines 2022. Secure Coding Guidelines for Java SE, Oracle. Retrieved from http://www.oracle.com/technetwork/java/seccodeguide-139067.html#6.
[87] Security Focus Vulnerability Database 2021. Security Focus Vulnerability Database, Accenture. Retrieved from https://www.securityfocus.com/.
[88] Security Vulnerabilities. 2017. Security Vulnerabilities (SQL Injection), MITRE Corporation. Retrieved from https://www.cvedetails.com/vulnerability-list/opsqli-1/sql-injection.html.
[89] Konrad Slind and Michael Norrish. 2008. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)*. 28–32. DOI: https://doi.org/10.1007/978-3-540-71067-7_6
[90] Stack Overflow 2008. Stack Overflow: A Q/A Site for Professional and Enthusiast Programmers, Stack Overflow. Retrieved from https://www.stackoverflow.com/.
[91] State of Software Security. 2016. State of Software Security, Veracode. Retrieved from https://www.veracode.com/sites/default/files/Resources/Reports/state-of-software-security-volume-7-veracode-report.pdf.
[92] Andreas Stefik and Susanna Siebert. 2013. An empirical investigation into programming language syntax. *Trans. Comput. Educ.* 13, 4 19 (Nov. 2013). DOI: https://doi.org/10.1145/2534973
[93] Jeffrey Stylos and Steven Clarke. 2007. Usability implications of requiring parameters in objects' constructors. In *International Conference on Software Engineering (ICSE'07)*. 529–539. DOI: https://doi.org/10.1109/ICSE.2007.92
[94] Symantec. 2017. Symantec Internet Security Threat Report. Retrieved from https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf.
[95] The Coq Development Team. 2017. Coq, v.8.7. Retrieved from https://coq.inria.fr.
[96] Philip S. Thomas, Bruno Castro da Silva, Andrew G. Barto, Stephen Giguere, Yuriy Brun, and Emma Brunskill. 2019. Preventing undesirable behavior of intelligent machines. *Science* 366, 6468 (22 Nov. 2019), 999–1004. DOI: https://doi.org/10.1126/science.aag3311
[97] Florian Tramer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, Jean-Pierre Hubaux, Mathias Humbert, Ari Juels, and Huang Lin. 2017. FairTest: Discovering unwarranted associations in data-driven applications. In *IEEE European Symposium on Security and Privacy (EuroS&P'17)*.
[98] Patricia A. Tun and Margie E. Lachman. 2006. Telephone assessment of cognitive function in adulthood: The brief test of adult cognition by telephone. *Age Ageing* 35, 6 (2006), 629–632.
[99] Tommi Unruh, Bhargava Shastry, Malte Skoruppa, Federico Maggi, Konrad Rieck, Jean-Pierre Seifert, and Fabian Yamaguchi. 2017. Leveraging flawed tutorials for seeding large-scale web vulnerability discovery. In *11th USENIX Workshop on Offensive Technologies (WOOT'17)*. USENIX Association. Retrieved from https://www.usenix.org/conference/woot17/workshop-program/presentation/unruh.
[100] Dirk van der Linden, Pauline Anthonysamy, Bashar Nuseibeh, Thein T. Tun, Marian Petre, Mark Levine, John Towse, and Awais Rashid. 2020. Schrödinger's security: Opening the box on app developers' security rationale. In *International Conference on Software Engineering (ICSE'20)*. DOI: https://doi.org/10.1145/3377811.3380394
[101] Daniel Votipka, Kelsey R. Fulton, James Parker, Matthew Hou, Michelle L. Mazurek, and Michael Hicks. 2020. Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It. In *29th USENIX Security Symposium (USENIX Security)*. Retrieved from https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-understanding.

[102] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek. 2018. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *IEEE Symposium on Security and Privacy (SP'18)*. 134–151. DOI: https://doi.org/10.1109/SP.2018.00003

[103] Zhiyuan Wan, Xin Xia, David Lo, Jiachi Chen, Xiapu Luo, and Xiaohu Yang. 2021. Smart contract security: A practitioners' perspective. In *International Conference on Software Engineering (ICSE'21)*.

[104] Sam Weber, Michael Coblenz, Brad Myers, Jonathan Aldrich, and Joshua Sunshine. 2017. Empirical studies on the security and usability impact of immutability. In *IEEE Cybersecurity Development (SecDev'17)*. 50–53. DOI: https://doi.org/10.1109/SecDev.2017.21

[105] Chamila Wijayarathna and Nalin Asanka Gamagedara Arachchilage. 2019. Using cognitive dimensions to evaluate the usability of security APIs: An empirical investigation. *Inf. Softw. Technol.* 115 (2019), 5–19. DOI: https://doi.org/10.1016/j.infsof.2019.07.007

[106] Jim Witschey, Shundan Xiao, and Emerson Murphy-Hill. 2014. Technical and personal factors influencing developers' adoption of security tools. In *ACM Workshop on Security Information Workers (SIW'14)*. 23–26. DOI: https://doi.org/10.1145/2663887.2663898

[107] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. 2015. Quantifying developers' adoption of security tools. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*. 260–271. DOI: https://doi.org/10.1145/2786805.2786816

[108] Shundan Xiao, Jim Witschey, and Emerson Murphy-Hill. 2014. Social influences on secure development tool adoption: Why security tools spread. In *ACM Conference on Computer Supported Cooperative Work & Social Computing (CSCW'14)*. 1095–1106. DOI: https://doi.org/10.1145/2531602.2531722

[109] Jing Xie, Heather Lipford, and Bei-Tseng Chu. 2012. Evaluating interactive support for secure programming. In *SIGCHI Conference on Human Factors in Computing Systems (CHI'12)*. 2707–2716. DOI: https://doi.org/10.1145/2207676.2208665

[110] Jing Xie, Heather Richter Lipford, and Bill Chu. 2011. Why do programmers make security errors? In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 161–164.

[111] Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning (ICML'19)*. Retrieved from http://proceedings.mlr.press/v97/yang19a/yang19a.pdf.

Received 1 December 2021; revised 29 June 2022; accepted 18 October 2022