Assessing Tool Support for Questions Developers Ask About Security Vulnerabilities

Justin Smith, Brittany Johnson, and Emerson Murphy-Hill
North Carolina State University
Raleigh, NC 27606

Bill Chu and Heather Richter Lipford
University of North Carolina at Charlotte
Charlotte, NC 28223

ABSTRACT

Program analysis tools should help developers answer the questions they ask while assessing the security of their code. To design effective tools, we must first understand which questions they ask. However, we lack a categorized catalog of such questions. We built this categorized catalog by conducting a laboratory experiment with novice and experienced software developers, observing their interactions with security vulnerabilities in iTrust, a Java medical records software system. The questions participants asked suggest that developers may lack adequate tool support. In particular, participants employed error-prone strategies while navigating from malicious data sources to sensitive data sinks and while managing all the tasks required to reason about a vulnerability.

1. INTRODUCTION

Software developers are a critical part of making software secure. When there is a security fault in a software system, it is up to the developers to determine the best way to remove the vulnerability and ensure proper function of the system in the future. There are a variety of ways developers can learn about, assess, and fix insecure code. It is particularly important to provide effective tools for secure systems, as security vulnerabilities are more likely to cause incidents that affect company profits as well as end users [3].

Toolsmiths produce static analysis tools to help developers detect potential security flaws in their code. One of example of such a tool is Find Security Bugs (FSB),[1] an extension of FindBugs.[2] FSB locates and reports on potential security vulnerabilities in software, such as SQL injection and cross-site scripting. Other tools, such as CodeSonar[3] and Coverity,[4] can also be used to detect and remove potential security vulnerabilities. Unfortunately, however, research suggests that many developers may not be using such static analysis tools, in part because tools do not always effectively communicate relevant information to developers [10].

[1]h3xstream.github.io/find-sec-bugs/
[2]findbugs.sourceforge.net
[3]grammatech.com/codesonar
[4]coverity.com/

Although existing work has cataloged developers' information requirements in a broad sense, including the notion that developers have questions about the security of their code [16], no prior study has specifically investigated questions that developers ask when using tools to diagnose and fix security vulnerabilities. As we show later in this paper, many questions are unique to using security tools. Furthermore, in non-security domains, work that identifies questions has helped toolsmiths improve the state-of-the-art of program analysis tools [13, 25, 27].

In this paper, we report a study conducted to create a better understanding of the information developers seek when using static analysis tools to identify and assess security vulnerabilities in code. We conducted a laboratory experiment with ten developers familiar with the iTrust software system.[5] We observed each developer as they assessed potential security vulnerabilities identified by FSB. We report the kinds of questions they asked while doing so, along with the strategies they used to answer them. Using a card sort methodology, we sorted the 559 questions into 17 categories.

Our work makes three contributions. First, we present a list of questions software developers ask when fixing security vulnerabilities. Second, based on our observations, we discuss the strategies developers use to answer their questions. Finally, we evaluate the existing tool support for answering security-related questions.

2. RELATED WORK

We have organized the related work into two subsections. Section 2.1 outlines the predominant approaches researchers use to evaluate security tools and Section 2.2 references similar studies that have explored the questions developers ask when modifying, understanding, or debugging their code.

2.1 Evaluation of Techniques

Developers use several techniques and tools to help secure their systems. Using a variety of metrics, many studies have assessed the effectiveness of the tools and techniques developers use to find and remove vulnerabilities from their code [1, 19, 20].

Much research has evaluated the effectiveness of tools and techniques based on their false positive rates and how many vulnerabilities they detect [1, 5, 11]. Jovanovic and colleagues attempt to address the problem of vulnerabilities in web applications by implementing and evaluating PIXY, a static analysis tool that detects cross-site scripting vulnerabilities in PHP scripts [11]. They

[5]http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=start

considered their tool effective because of its low false positive rate (50%) and its ability to find vulnerabilities previously unknown. Similarly, Livshits and Lam evaluated their own approach to static analysis for detecting security vulnerabilities; the static analyzers used are created from specifications provided by the user [19]. They also found their tool to be effective because it had a low false positive rate.

Austin and Williams compared the effectiveness of four existing techniques for discovering security vulnerabilities: systematic and exploratory manual penetration testing, static analysis, and automated penetration testing [1]. Comparing the four approaches based on number of vulnerabilities found, false positive rate, and efficiency, they reported that no one technique was capable of discovering every type of vulnerability. Dukes and colleagues conducted a case study comparing static analysis and manual testing techniques used to find security vulnerabilities [5]. They found combining manual testing and static analysis was most effective, because it located the most vulnerabilities.

These studies use various measures of effectiveness, such as false positive rates or vulnerabilities found by a tool, but none focus on how developers interact with the tool. Further, they do not evaluate whether the tools under study answer questions developers have about the code or vulnerabilities.

2.2 Answering Developer Questions

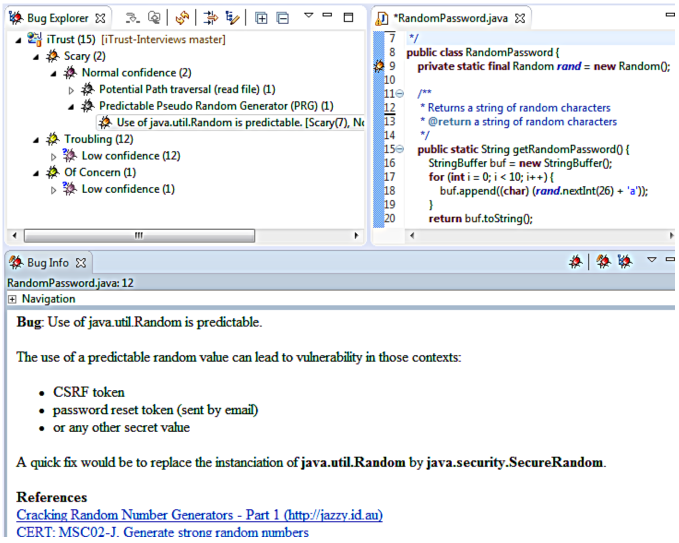
Several studies have explored the knowledge requirements of developers when coding. Similar to our work, some existing studies focus on the questions developers ask to determine these knowledge requirements [15, 16]. In contrast, our study focuses specifically on the knowledge requirements of developers when assessing security vulnerabilities.

Much of the work on answering developer questions has occurred in the last decade. LaToza and Myers surveyed professional software developers to understand the questions developers ask during their daily coding activities, focusing on the hard to answer questions [16]. Futhermore, after observing developers in a lab study, they discovered that the questions developers ask tend to be questions revolving around searching through the code for target statements, or reachability questions [15]. Ko and Myers developed WHYLINE, a tool meant to ease the process of debugging code by helping answer "why did" and "why didn't" questions. [12]. They found that developers were able to complete more debugging tasks when using their tool than they could without it. Fritz and Murphy developed a model and prototype tool to assist developers with answering the questions they want to ask based on interviews they conducted [6].

Our work builds on the work of LaToza and Myers, which found that some of the hard-to-answer questions developers have surround the implications of code changes on the security of their code.

3. METHODOLOGY

We conducted a laboratory experiment with ten software developers. In our analysis, we extracted and categorized the questions developers asked during each study session. Section 3.1 outlines the research questions we sought to answer. Section 3.2 details how the study was designed and Section 3.3 describes how we performed



Bug Explorer
iTrust (15) [iTrust-Interviews master]
Scary (2)
Normal confidence (2)
Potential Path traversal (read file) (1)
Predictable Pseudo Random Generator (PRG) (1)
Use of java.util.Random is predictable. [Scary(7), N
Troubling (12)
Low confidence (12)
Of Concern (1)
Low confidence (1)
*RandomPassword.java
Bug Info
RandomPassword.java: 12
Navigation

```
 */
public class RandomPassword {
   private static final Random rand = new Random();

   /**
    * Returns a string of random characters
    * @return a string of random characters
    */
   public static String getRandomPassword() {
      StringBuffer buf = new StringBuffer();
      for (int i = 0; i < 10; i++) {
         buf.append((char) (rand.nextInt(26) + 'a'));
      }
      return buf.toString();
```

Bug: Use of java.util.Random is predictable.

The use of a predictable random value can lead to vulnerability in those contexts:

- CSRF token
- password reset token (sent by email)
- or any other secret value

A quick fix would be to replace the instanciation of **java.util.Random** by **java.security.SecureRandom**.

References
Cracking Random Number Generators - Part 1 (http://jazzy.id.au)
CERT: MSC02-J. Generate strong random numbers

Figure 1: The study environment.

data analysis. Experimental materials can be found online. [6]

3.1 Research Questions

We want to answer the following research questions:

- **RQ1**: What types of security related questions do developers ask?
- **RQ2**: How do current tools and approaches support developers in answering these questions?

3.2 Study Design

Each session started with a five-minute briefing section, which included time for participants to familiarize themselves with the study environment and FSB. Following the briefing period, participants progressed through encounters with four vulnerabilities. Figure 1 depicts the configuration of the integrated development environment (IDE) for one of these encounters. All participants consented to participate in our study, which had institutional review board approval, and to have their session recorded using screen and audio capture software. Finally, each session concluded with several demographic and open-ended discussion questions.

3.2.1 Materials

Participants used Eclipse to explore vulnerabilities in iTrust, an open source Java medical records web application that ensures the privacy and security of patient records according to the HIPAA statute.[7] During the briefing session, participants were given time to familiarize themselves with the development environment and ask any questions about the experimental setup. Participants were equipped with a version of FindBugs extended with FSB. We chose FSB because of its low barriers to adoption; it is free, open source, easy to install and configure, and contained within the integrated development environment (IDE). FSB is also similar to other more sophisticated static analysis tools — such as those listed by NIST,[8]

[6]`http://www4.ncsu.edu/~bijohnso/security-questions.html`
[7]`hhs.gov/ocr/privacy/`
[8]`samate.nist.gov/index.php/Source_Code_Security_Analyzers.html`

Table 1: Demographics of study participants

Participant	Job Title	Vulnerability Familiarity	Experience Years
P1	Student	★★⯪☆☆	4.5
P2	Test Engineer	★★★☆☆	8
P3	Development Tester	★★☆☆☆	6
P4	Software Developer	★★☆☆☆	6
P5	Student	★★★★☆	10
P6	Student	★☆☆☆☆	4
P7	Software Developer	★★★★☆	4.5
P8	Student	★★★☆☆	7
P9	Software Consultant	★★★☆☆	5
P10	Student	★★★☆☆	8

OWASP,[9] and WASC[10] — in terms of the types of security vulnerabilities it detects.

3.2.2 *Participants*

To simulate a more realistic work environment, participants were required to have significant experience working on the iTrust code base. All participants either completed or served as teaching assistants for a semester-long software engineering course that focused on developing iTrust.

For our experiment, we recruited ten software developers, five students and five professionals. Participants ranged in programming experience from 4 years to 10 years, averaging 6.3 years. All interviews were conducted in-person. We recruited participants from personal contacts and class rosters, using snowball sampling to find additional qualified participants. Since extracting data from each interview required intensive analysis, we ceased recruitment when we determined that we had reached saturation and few new questions were emerging from each additional interview.

Table 1 gives additional demographic information on each of the ten participants. Participants self-reported their familiarity with security vulnerabilities on a 5 point Likert scale. In the remainder of this paper, we will refer to participants by the abbreviations found in the participant column of the table. Since the focus of our study is to identify the questions developers ask independent of their background, we report on experience to ensure a degree of diversity in our sample and contextualize participants' responses.

3.2.3 *Tasks*

Each participant was asked to assess four vulnerabilities. We presented each participant with this number of vulnerabilities, because in preliminary pilot sessions, participants spent approximately 10-15 minutes with each vulnerability and showed signs of fatigue after 60 minutes.

When selecting tasks, we gave preference to existing iTrust vulnerabilities and sought a broad set of question topics. FSB uses topic abbreviations to group similar error messages. For example the abbreviation for the topic Path Traversal is PT and applies to notifications that warn about potential path traversal. FSB reports on several types of vulnerabilities. To cover a broader set of question topics, we ensured each vulnerability pertained to a different topic. FSB identified three types of naturally occurring vulnerabilities: cross-site scripting, path traversal, and predictable random vulnerabilities. We inserted a SQL injection vulnerability by making minimal alterations to one of the database access objects. Our modification preserved the functionality of the original code and was based on examples of SQL injection found on OWASP and in open-source projects. Table 2 summarizes each of the four vulnerabilities.

During the briefing section, we asked participants to pretend they are in charge of security for iTrust and to approach the vulnerabilities as if they were in their normal work environment. Additionally, we asked them to use a think-aloud protocol, which encourages the participant to verbally announce their thought process as they complete a task or activity [22]. Specifically, they were asked to, "Say any questions or thoughts that cross your mind regardless of how relevant you think they are." We recorded both audio and the screen as experimental artifacts for data analysis.

3.3 Data Analysis

We analyzed session data using an approach inspired by grounded theory [8]. First, we transcribed all the audio-video files using oTranscribe.[11] Each transcript, along with the associated recording, was analyzed by two of the authors for implicit and explicit questions. The two question sets for each session were then iteratively compared against each other until the authors reached agreement on the question sets. In the remainder of this section, we will detail the question extraction process and question sorting processes, including the criteria used to determine which statements qualified as questions.

3.3.1 *Question Criteria*

Participants ask both explicit and implicit questions. Drawing from previous work on utterance interpretation, we developed five criteria to assist in the uniform classification of participant statements [18]. A statement was coded as a question only if it met one of the following criteria:

- **The participant explicitly asks a question.**
  Example: *Why aren't they using `PreparedStatements`?*
- **The participant makes a statement and explores the validity of that statement.**
  Example: *It doesn't seem to have shown what I was looking for. Oh, wait! It's right above it...*
- **The participant uses key words such as, "I assume," "I guess," or "I don't know."**
  Example: *I don't know that it's a problem yet.*
- **The participant clearly expresses uncertainty over a statement.**
  Example: *Well, it's private to this object, right?*
- **The participant clearly expresses a knowledge requirement by describing plans to acquire information.**
  Example: *I would figure out where it is being called.*

3.3.2 *Question Extraction*

Using the criteria outlined in the previous section, two of the authors independently coded each session. When we identified an

[9]`owasp.org/index.php/Source_Code_Analysis_Tools`
[10]`projects.webappsec.org/w/page/61622133/StaticCodeAnalysisList`
[11]`otranscribe.com`

Table 2: Four vulnerability exploration tasks

Vulnerability	Short Description	Severity Rank
Potential Path Traversal	An instance of java.io.File is created to read a file.	"Scary"
Predictable Random	Use of java.util.Random is predictable.	"Scary"
Servlet Parameter	The method getParameter returns a String value that is controlled by the client.	"Troubling"
SQL Injection	[Method name] passes a non-constant String to an execute method on an SQL statement.	"Of Concern"

explicit question or implicit question, which is a statement that satisfies one or more of the criteria, we marked the transcript, highlighted the participant's original statement, and clarified the question being asked. Question clarification typically entailed rewording the question to best reflect the information the participant is trying to acquire. From the ten sessions, the first coder extracted 421 question statements; the other coder extracted 389.

To make sure our list of questions was exhaustive, two independent reviewers coded the transcripts using the criteria. It was sometimes difficult to determine what statements should be extracted as questions; the criteria helped ensure both reviewers only highlighted the statements that reflected actual questions.

3.3.3 *Question Review*

To remove duplicates and ensure the validity of all the questions, each transcript was reviewed jointly by the two authors that initially coded it. During the second pass, the two reviewers examined each question statement, discussing the justification for each question based on the previously stated criteria. The two reviewers merged duplicate questions, favoring the wording that was most strongly grounded in the experiment artifacts. This process resulted in a total of 559 questions.

Each question that was only identified by one author required verification. If the other author did not agree that such a question met at least one of the criteria, the question was removed from the question set and counted as a disagreement. The reviewers were said to agree when they merged a duplicate or verified a question. Depending on the participant, inter-reviewer agreement ranged from 91% to 100%. Across all participants, agreement averaged to 95%. The agreement scores suggest that the two reviewers consistently held similar interpretations of the question criteria.

3.3.4 *Question Sorting*

To organize our questions and facilitate discussion, we performed an *open* card sort [9]. Card sorting is typically used to help structure data by grouping related information into categories. In an *open* sort, the sorting process begins with no notion of predefined categories. Rather, sorters derive categories from emergent themes in the cards.

We performed our card sort in three distinct stages: clustering, categorization, and validation. In the first stage, we formed question clusters by grouping questions that identified the same information requirements. In this phase we focused on rephrasing similar questions and grouping duplicates. For example, P1 asked, *Where can I find information related to this vulnerability?* P7 asked, *Where can I find an example of using `PreparedStatements`?* and P2 asked, *Where can I get more information on path traversal?* Of these questions, we created a question cluster labeled *Where can I get more information?* At this stage, we discarded five unclear or non pertinent questions and organized the remaining 554 into 155 unique question clusters.

In the second stage, we identified emergent themes and grouped the clusters into categories based on the themes. For example, we placed the question *Where can I get more information?* into a category called ***Resources/Documentation***, along with questions like *Is this a reliable/trusted resource?* and *What information is in the documentation?* Table 3 contains the 17 categories along with the number of distinct clusters each contains.

To validate the categories that we identified, we asked two independent researchers to sort the question clusters into our categories. Rather than sort the entire set of questions, we randomly selected 43 questions for each researcher to sort. The first agreed with our categorization with a Cohen's Kappa of $\kappa = .63$. Between the first and second researcher we reworded and clarified some ambiguous questions. The second researcher exhibited greater agreement ($\kappa = .70$). These values are within the $.60 - .80$ range, indicating substantial agreement [14].

4. RESULTS

4.1 Interpreting the Results

In the next four sections, we discuss our experiment's results using the categories we described in the previous section. Due to their large number, we grouped the categories to organize and facilitate discussion about our findings. Table 3 provides an overview of this structure. For each category, we selected several questions to discuss. A full list of distinct questions can be found online.[12] The numbers next to the category titles denote the number of participants that asked questions in that category and the total number of questions in that category — in parenthesis and brackets respectively. Similarly, the number in parenthesis next to each question marks the number of participants that asked that question.

When discussing the questions participants asked for each category, we will use phrases such as "*X* participants asked *Y*." Note that this work considers a relatively small sample and is exploratory and qualitative in nature. Though we present information about the number of participants who ask specific questions, the reader should not infer any quantitative generalizations.

4.2 Vulnerabilities, Attacks, and Fixes, Oh My!

4.2.1 **Understanding Alternative Fixes and Approaches (8){11}**

When resolving security vulnerabilities, participants explored alternative ways to achieve the same functionality more securely. For example, when evaluating the SQL Injection vulnerability, participants found resources that suggested using the `PreparedStatement` Class instead of Java `Statement` Class.

Does the alternative function the same as what I'm currently using? (6)
What are the alternatives for fixing this? (4)
Are there other considerations to make when using the alternative(s)? (3)

[12] `www4.ncsu.edu/~bijohnso/questions.pdf`

Table 3: Organizational Groups and Emergent Categories

Group	Category	Clusters	Location in Paper
Vulnerabilities, Attacks, and Fixes, Oh My!	Understanding Alternative Fixes and Approaches	11	Section 4.2.1
	Preventing and Understanding Potential Attacks	11	Section 4.2.2
	Assessing the Application of the Fix	9	Section 4.2.3
	Relationship Between Bugs	3	Section 4.2.4
Code and the Application	Locating Information	11	Section 4.3.1
	Control Flow and Call Information	13	Section 4.3.2
	Data Storage and Flow	11	Section 4.3.3
	Code Background and Functionality	17	Section 4.3.4
	Application Context/Usage	9	Section 4.3.5
	End-User Interaction	3	Section 4.3.6
Individuals	Developer Planning and Self-Reflection	14	Section 4.4.1
	Understanding Concepts	6	Section 4.4.2
	Confirming Expectations	1	Section 4.4.3
Problem Solving Support	Bug Severity and Rank	4	Section 4.5.3
	Understanding and Interacting with Tools	9	Section 4.5.2
	Resources and Documentation	10	Section 4.5.1
	Error Messages	3	Section 4.5.4
	Uncategorized	10	

How does my code compare to the alternative code in the example I found? (2)
Why should I use this alternative method/approach to fix the vulnerability? (2)

Observations. Eight participants had questions that fit into this category. The FSB notifications for the SQL Injection and Predictable Random vulnerabilities explicitly offer alternative fixes. As noted, the message associated with the SQL Injection vulnerability suggests switching to use `PreparedStatements`. The message associated with the Predictable Random vulnerability suggests switching to use `java.security.SecureRandom`. In other cases, participants turned to a variety of sources, such as StackOverflow, official documentation, and personal blogs for alternative approaches. Three participants specifically mentioned StackOverflow as a source for better understanding alternative approaches and fixes. P7 preferred StackOverflow as a resource, because it included real-world examples and elaborated on what was wrong with the code as it was currently written. While attempting to assess the Servlet Parameter vulnerability (Table 2), P8 decided to explore some resources on the web and came across a resource that appeared to be affiliated with OWASP[13]. Because he recognized OWASP as "the authority on security," he clicked the link and used it to make his final decision regarding the vulnerability. Despite the useful information some participants found, often the candidate alternative did not readily provide meta-information about tradeoffs or the process of applying suggestions. For example, P9 found a suggestion on StackOverflow that he thought might work, but it was not clear if it could be applied to the code in iTrust.

4.2.2 Preventing and Understanding Potential Attacks (10){11}

Unlike other types of code defects that may cause code to function unexpectedly or incorrectly, security vulnerabilities expose the code to potential attacks. For example, the Servlet Parameter vulnerability introduced the possibility of SQL injection, path traversal, command injection, and cross-site scripting attacks.

Is this a real vulnerability? (7)
What are the possible attacks that could occur? (5)
Why is this a vulnerability? (3)
How can I prevent this attack? (3)
How can I replicate an attack to exploit this vulnerability? (2)
What is the problem (potential attack)? (2)

[13] `owasp.org/index.php/Main_Page`

Observations. Participants sought information about the variety of attacks that could occur in a given context. To that end, five participants asked, *What are the possible attacks that could occur?* For example, within the first minute of his analysis, P2 stated that he was thinking about the different types of attacks that could target web applications, such as iTrust. He simultaneously wondered which types of attacks were possible and which attacks he might need to prevent.

Participants also sought more specific attack-information about particular attacks as they were being considered. Participants hypothesized about specific attack vectors, how to execute those attacks, and how to prevent those attacks now and in the future. Seven participants asked the question, *Is this a real vulnerability?* To answer that question, participants searched for hints that an attacker could actually execute a given attack in a specific context. For example, P10 determined that the Predictable Random vulnerability was "real" because an attacker could deduce the random seed and use that information to determine other users' passwords.

4.2.3 Assessing the Application of the Fix (9){9}

Once participants had identified a specific approach for fixing a security vulnerability (Section 4.2.1), they asked questions about applying the fix to the code. For example, when considering the use of `java.security.SecureRandom` to resolve the Predictable Random vulnerability, participants questioned the applicability of the fix and the consequences of making the change. The questions in this category differ from those in ***Understanding Alternative Fixes and Approaches*** (Section 4.2.1). These questions focus on the process of applying and reasoning about a given fix, rather than identifying and understanding possible fixes.

Will the error go away when I apply this fix? (5)
How do I use this fix in my code? (4)

How do I fix this vulnerability? (4)
How hard is it to apply a fix to this code? (3)
Is there a quick fix for automatically applying a fix? (2)
Will the code work the same after I apply the fix? (2)
What other changes do I need to make to apply this fix? (2)

Observations. When searching for approaches to resolve vulnerabilities, participants gravitated toward fix suggestions provided by the notification. The notifications associated with the Predictable Random vulnerability and the SQL Injection vulnerability both provided fix suggestions. All participants proposed solutions that involved applying one or both of these suggestions. Specifically, P2 commented that it would be nice if all the notifications contained fix suggestions.

However, unless prompted, none of the participants commented on the disadvantages of using fix suggestions. While exploring the Predictable Random vulnerability, many participants, including P1, P2, and P6 decided to use `java.security.SecureRandom` without considering any alternative solutions, even though the use of that suggested fix reduces performance.

4.2.4 Relationship Between Bugs (4){3}

Some participants asked questions about the connections between co-occurring vulnerabilities and whether similar vulnerabilities exist elsewhere in the code. For example, when participants reached the third and fourth vulnerabilities, they began noticing and speculating about the similarities between the vulnerabilities they inspected.

Are all the bugs related in my code? (3)
Does this other piece code have the same bug as the code I'm working with? (1)

4.3 Code and the Application

4.3.1 Locating Information (10){11}

Participants asked questions about locating information in their coding environments. In the process of investigating vulnerabilities, participants searched for information across multiple classes and files. Unlike Sections 4.3.2 and 4.3.3, questions in this category refer more generally to the process of locating information, not just about locating calling information or data flow information.

Where is this used in the code? (10)
Where are other similar pieces of code? (4)
Where is this method defined? (1)

Observations. All ten participants wanted to locate where defective code and tainted values were being used throughout the system. Most of these questions occurred in the context of assessing the Predictable Random vulnerability. Specifically, participants wondered where the potentially insecure random number generator was being used.

In other cases, four participants wanted to find parts of the code that implemented similar functionality to the code they were inspecting. They hypothesized that other parts of the code implemented the same functionality using more secure patterns. For example, while assessing the SQL Injection vulnerability, P2 and P5 both wanted to find other modules that created SQL statements. All participants completed this task manually by scrolling through the package explorer and searching for code using their knowledge of the application.

4.3.2 Control Flow and Call Information (10){13}

Developers seek information about which methods that get called, and whether they may get called with tainted values.

Where is the method being called? (10)
How can I get calling information? (7)
Who can call this? (5)
Are all calls coming from the same class? (3)
What gets called when this method gets called? (2)

Observations. In particular, participants asked some of these questions while exploring the Path Traversal vulnerability. The private method containing this vulnerability was only called by one method, `parseAndCache()`, which was defined directly above it. `parseAndCache()`, which was also private, was called by a public method. Tracing up the chain, the method containing the vulnerability was eventually called from multiple classes that were all contained within a test package.

All ten participants wanted call information for this vulnerability, often asking the question, *Where is this method being called?* However, participants used various strategies to obtain the same information. The most basic strategy was simply skimming the file for method calls. which was error-prone because participants could easily miss calls. Other participants used the Eclipse's CODE HIGHLIGHTER tool, which, to a lesser extent, was error-prone for the same reason. Further, it only found calls within the current file. Participants additionally employed Eclipse's TEXT SEARCH tool, which found all occurrences of a method name, but there was no guarantee that strings returned referred to the same method. Also, it returned references that occurred in dead code or comments. Alternatively, Eclipse's FIND REFERENCES tool identifies proper references to a single method. Unlike the previous strategies, Eclipse's CALL HIERARCHY tool enables users to traverse a project's entire call structure. That said, it does not identify calls made from framework hooks or APIs. Participants also faced tool awareness issues, a common blocker for both novice and experienced users [21].

Eventually, many participants hypothesized that all the calls might originate from the same (test) class. Three participants explicitly asked, *Are all calls coming from the same class?* To verify this assumption, participants primarily used the heuristic of visually examining the list of calling methods and their package names, scanning for the "test" keyword. This strategy does not scale particularly well and would fail if the methods were named incorrectly, or if they accidentally missed one. Howver, in this case, these participants hypothesized correctly that the vulnerability was a false positive in the sense that the code was only called from test classes.

4.3.3 Data Storage and Flow (10){11}

Participants also asked questions about the data being stored in and carried throughout the program. Participants often wanted to understand the type of data being collected and stored, where the data came from, and where it was going. For example, participants wanted to determine whether data was generated by the application or passed in by the user. They also wondered if the data was ever being stored in a database.

Where does this information/data go? (9)
Where is the data coming from? (5)
How is data put into this variable? (3)
Does data from this method/code travel to the database? (2)
How do I find where the information travels? (2)

How does the information change as it travels through the programs? (2)

Observations. Participants asked questions about the data pipeline when assessing three of the four vulnerabilities, many of these questions arose while assessing the Path Traversal vulnerability.

While exploring this vulnerability, participants adapted tools such as the CALL HIERARCHY tool to also explore the program's data flow. Specifically, participants used the CALL HIERARCHY tool in combination with manual searching. As we discussed in ***Control Flow and Call Information***, the call hierarchy tool helped participants locate callers to the methods containing vulnerable code. For instance, once participants like P6 located the caller, they manually searched for the locations where data was being manipulated within the method.

4.3.4 Code Background and Functionality (9){17}

Participants asked questions concerning the background and intended function of the code being analyzed. The questions in this category differ from those in Section 4.3.5, because they focus on the lower level implementation details of the code.

What does this code do? (9)
Why was this code written this way? (5)
Why is this code needed? (3)
Who wrote this code? (2)
Is this library code? (2)
How much effort was put into this code? (1)

Observations. Participants were interested in how the code was written as well as the history of the code. For example, they asked why it was written a specific way, who wrote the code, or how much effort had been put into the code. P2 asked about the effort put into the code to determine whether he trusted that the code was written securely. While assessing the Potential Path Traversal vulnerability, some participants expressed interest in consulting the code's author for help. For example, P10 stated that he would normally ask his team members for background on how the code worked.

4.3.5 Application Context/Usage (9){9}

Unlike questions in Section 4.3.4, these questions refer to system-level concepts. For instance, often when assessing the vulnerabilities, participants wanted to know what the code was being used for, whether it be testing, creating appointments with patients, or generating passwords.

What is the context of this bug/code? (4)
Is this code used to test the program/functionality? (4)
What is the method/variable used for in the program? (3)
Will usage of this method change? (2)
Is the method/variable ever being used? (2)
Are we handling secure data in this context? (1)
How does the system work? (1)

Observations. Four participants wanted to know the context in which the code being assessed was used or the context of the bug being assessed. Similarly, three participants found themselves trying to determine what a specific method or variable is used for in the application. Both of these questions came up for three of the four vulnerabilities.

P7 had questions concerning the context of the code associated with the Potential Path Traversal and Predictable Random vulnerabilities. For the former vulnerability, he wanted to know the general context in which the code is used; for the latter, he wanted to know the legal context in which the code is being used. P2, when thinking about how he might determine the context in which the code associated with the Predictable Random vulnerability is used, stated that the tool would have been more useful if it could tell him this information. He followed this statement with the notion that this sort of tool may be "unrealistic."

Participants also tried to determine if the code associated with the vulnerability was used to test the system. P2, P4, P9, and P10 asked whether the code they were examining occurred in classes that were only used to test the application. To answer this question, participants sometimes used tools for traversing the call hierarchy; using these types of tools allowed them to narrow their search to only locations the code of interest is being called. Compared to the tool support provided for determining general context, participants could answer this question relatively easily using existing tool support.

4.3.6 End-User Interaction (8){3}

Questions in this category deal with how end users might interact with the system or a particular part of the system. Some participants wanted to know whether users could access critical parts of the code and if measures were being taken to mitigate potentially malicious activity. For example, while assessing the Potential Path Traversal vulnerability, participants wanted to know whether the path is sanitized somewhere in the code before it is used.

Is there input coming from the user? (4)
Does the user have access to this code? (4)
Does user input get validated/sanitized? (4)

Observations. Eight participants asked questions regarding the level of interaction end-users have with the code of interest. Though none of these questions were asked more than others, some questions co-occurred for a given vulnerability. For example, when assessing the Potential Path Traversal vulnerability, P1 and P6 wanted to know if the input was coming from the user along with whether the input was being validated in the event that the input did come from the user. For these participants, finding the answer required manual inspection of surrounding and relevant code. For instance, P6 found a `Validator` method, which he manually inspected, to determine if it was doing input validation.

When assessing the Potential Path Traversal vulnerability, four participants asked whether end-users had access to the part of the code being analyzed. P2 used CALL HIERARCHY to answer this question by tracking where the method the code is contained in gets called; for him, the vulnerability only existed if the user had access to the code. P1 and P6 went on a similar mission and determined that because all the calls to the code of interest appeared to happen in methods called `testDataGenerator()`, the code should be fine as it is written. Though participants found the answers to their questions, it took time for them to get the answer using tools designed to answer other questions.

4.4 Individuals

4.4.1 Developer Planning and Self-Reflection (8){14}

All of the questions in this category involve the developer's thoughts or the individual's relationship to the problem, rather than specifics of the code or the error notification.

Do I understand? (3)
What should I do first? (2)
What was that again? (2)
Is this worth my time? (2)
Why do I care? (2)
Have I seen this before? (1)
Where am I in the code? (1)

Observations. Eight participants asked questions that were meant to be plan-oriented or reflective. These questions occurred in all four vulnerabilities. Participants most frequently asked if they understood the entirety of the situation. This question was typically followed by looking back at the error message details. For instance, as P6 started exploring the validity of the SQL Injection vulnerability, he wanted to know if he fully understood the error message before he started exploring, so he went back to reread the error message before investigating further.

4.4.2 Understanding Concepts (7){6}

Some participants encountered unfamiliar terms and concepts in the code and vulnerability notifications. For instance, while parsing the potential attacks listed in the notification for the Servlet Parameter vulnerability, some participants did not know what a CSRF token was.

What is this concept? (6)
How does this concept work? (4)
What is the term for this concept? (2)

Observations. Seven participants had questions regarding the concepts and terms relevant to the potential vulnerability under consideration. Questions in this category appeared at least once for each vulnerability. Six of the seven participants who had questions in this category asked the question *What is this concept?* for at least one of the potential vulnerabilities.

Participants often clicked links leading to more information. For example, while assessing the Potential Path Traversal vulnerability, P2, unsure of what path traversal was, clicked the link labeled "path traversal attack" provided by FSB to get more information. If a link was not available, they went to the web to get more information or noted that the notification could have included more information on those concepts. While parsing the information provided for the Predictable Random vulnerability, P7 and P8 did not know what CSRF token was. The notification for this vulnerability did not include links so they searched the web for more information. When asked what information he would like to see added to the notification for the Servlet Parameter vulnerability, which also did not include any links, P4 noted he wold have liked the notification to include what a servlet is and how it related to client control.

4.4.3 Confirming Expectations (4){1}

A few participants found themselves in situations where they wanted to be able to confirm whether the code is doing what they expect. The distinct question asked in this category was, *Is this doing what I expect it to?*

4.5 Problem Solving Support

4.5.1 Resources and Documentation (10){10}

Many participants indicated they would use outside resources and documentation to decide how to proceed with a given vulnerability. For example, when assessing the Potential Path Traversal vulnerability, participants wanted to know what their team members would do or if they could provide any additional information about the vulnerability.

Can my team members/resources provide me with more information? (5)
Where can I get more information? (5)
What information is in the documentation? (5)
How do resources do to prevent or resolve this? (5)
Is this a reliable/trusted resource? (3)
What type of information does this resource link me to? (2)

Observations. All ten participants had questions regarding the resources and documentation available to help them assess a given vulnerability. Even with the links to external resources provided by two of the vulnerabilities, participants still had questions about available resources. Some participants used the links provided by FSB to get more information about the vulnerability. Participants who did not click the links in the notification had a few reasons for not doing so. For some participants, the hyperlinked text was not descriptive enough to know what information the tool was offering; others did not know if they could trust the information they found.

Some participants found themselves clicking the links provided by FSB thinking they would find what they are looking for, to find that they actually had not. For example, P2 clicked the first link on path traversal while trying to understand the Potential Path Traversal vulnerability hoping to find information on how to resolve the vulnerability. When he did not see that information, he had to go back and attempt another web search for that information. A few participants did not know the links existed, so they typically either searched the web or noted other strategies they might use, such as asking other developers on the project.

4.5.2 Understanding and Interacting with Tools (8){9}

Throughout the study participants interacted with a variety of tools including FSB, CALL HIERARCHY, and FIND REFERENCES. While interacting with these tools, participants asked questions about how to access specific tools, how to properly use the tool, and how to interpret its output.

Why is the tool complaining? (3)
Can I verify the information the tool provides? (3)
What is the tool's confidence? (2)
What is the tool output telling me? (1)
What tool do I need for this? (1)

Observations. Some of the questions participants asked deal with having access to or knowing how to use the tools needed to complete a certain task. Participants sometimes found themselves in situations where they needed information but could not determine how to invoke the tool or possibly did not know what tool they would use at all. This seems to indicate a tool awareness or education problem.

Other questions dealt with the presentation of information provided by the tool. Participants wanted to be able to explore things about the tool's output, and typically had to manually determine the answers to their questions. For example, FSB reports its confidence on the potential defect in the notification. However, some participants found it difficult to process this information.

4.5.3 Bug Severity and Ranking (5){4}

FSB estimates the severity of each vulnerability it encounters and reports those rankings to its users (Table 2). Participants asked questions while interpreting these rankings.

How serious is this bug? (2)
How do the rankings compare? (2)
What do the bug rankings mean? (2)

Observations. Most of these questions came from participants wanting to know more about the tool's method of ranking the vulnerabilities in the code. Participants who attempted to interpret the tool's severity and ranking metrics often had difficulty understanding what they meant. For example, when assessing the Potential Path Traversal vulnerability, P1 discovered the rank, severity, and confidence information provided by the tool. He noted that it would be helpful and added it to his assessment process for the following vulnerabilities. As he began working through the final vulnerability (SQL Injection), he realized he may not have as good of an understanding of the tool's metrics as he thought and did a web search for clarification. P6, among others, did not notice the ranking and severity information provided by the tool.

4.5.4 Error Messages (6){3}

FSB provided long and short descriptions of each vulnerability (Figure 1). Participants read and contemplated these notifications to guide their analysis.

What does the error message say? (5)
What is the relationship between the error message and the code? (2)
What code caused this error message to occur? (2)

Observations. Six participants had questions about the vulnerability notifications; five of these participants asked, *What does the error message say?*, a typical question for someone to ask when using tools to analyze source code. More interestingly, they also asked questions about how to relate information contained in the error message back to the code. For example, the Predictable Random vulnerability notes that a predicable random value could lead to an attack when being used in a secure context. Many participants attempted to relate this piece of information back to the code by looking to see if anything about the code that suggested it is in a secure context. In this situation, the method containing the vulnerability was named `randomPassword()`, which suggested to participants that the code was in a secure context and therefore a vulnerability that should be resolved.

5. DISCUSSION

Many of the questions participants asked related specifically to software security. In particular, questions in Sections 4.2.2, 4.3.6, and 4.5.3 are most directly connected to security activities. However, participants also asked questions that relate to more general bug finding tools. In fact, some such questions have been identified in other research contexts [16]. In this section we will contextualize both these types of question, focusing on the failure points of certain question-answering strategies and how tools can support more successful strategies.

5.1 Flow Navigation

When performing sensitive operations, developers want to trace incoming data to determine if it originated from a malicious source. Similarly, given data from the user, developers are interested in determining how it gets used in an application and trace whether it passes through the correct sanitization steps before being used in a sensitive operation. Questions related to these tasks appear in four different categories (Sections 4.3.1, 4.3.2, 4.3.3, 4.3.6).

We observed developers using several strategies to answer questions related to these tasks, including: tracing up and down method calls to determine if the source is user-facing or if data is being used in sensitive operations; looking at class names for known user-facing or non user-facing classes; using existing knowledge of sensitive operations and data sources to reason about the code without searching. A mix of existing lightweight (i.e. text search and code highlighting) and heavyweight tools (i.e. CALL HIERARCHY and FIND REFERENCES) partially support some of these strategies.

However, even with some degree of tool support, the strategies we observed exhibit several limitations. First, lightweight tools that help developers navigate the flow of a program often require multiple invocations and may return partial or irrelevant information. Existing heavyweight tools provide more complete and precise analysis, but are undermined by barriers to invocation [10]. Even still, existing sophisticated tools still may return incomplete information. For example, CALL HIERARCHY does not alert developers when methods are being called from outside the system (i.e. framework hooks and api calls), and does not allow developers to trace the call structure while also tracking where specific parameters are set. Finally, strategies that rely on developers' prior knowledge of the code base are vulnerable to several failures. For example, if developers are unaware that they are working with unstable recently-changed code they may make incorrect assumptions based on a false sense of knowledge [7]. Additionally, when a developer is only responsible for a portion of the system, as is often the case CITE HERE, he may be unable to reason about the system-wide implications of a change.

We are not the first to suggest that developers could benefit from increased support in navigating program flow. Researchers have advanced promising tools that help developers visualize call graphs [17] and trace control flow to its origin [2]. However, in a security context, developers are particularly interested in how data propagates throughout a program and ask questions about data flow while navigating the call hierarchy (Sections 4.3.3 and 4.3.2). Perhaps a new tool could help developers reason about control flow and data flow simultaneously by leveraging the strengths of existing tools. Such a tool could adapt a precise, yet heavyweight, control flow navigation tool by reducing its barriers to invocation, a technique successfully demonstrated by SPYGLASS [26]. Further, the tool could allow developers to specify specific parameters to trace and display information about where those values are modified. Finally, the tool could alert developers to the possibility that some calls might originate from outside the system and that tainted user data may enter the system through other channels.

5.2 Vulnerability Assistant

Static analysis tools help developers locate potential vulnerabilities and also provide explanatory notifications. To completely resolve potential vulnerabilities, developers must perform many cognitively demanding tasks beyond simply locating the vulnerability and understanding the notification, as is evidenced by the breadth of categories we identified. To determine whether a vulnerability is a true positive and if so how to fix it, we observed developers deploying a mix of several high-level strategies including: inspecting the code; navigating to other relevant areas of the code; comparing the vulnerability to previous vulnerabilities; consulting teammates, documentation, and other resources; weighing existing knowledge against information in the notification; and reasoning about the feasibility of all the possible attacks.

However, static analysis tools that help developers locate vulnerabilities often provide inadequate support for these strategies that developers use to resolve vulnerabilities. Some of the failures that we observed include not supporting navigating the code base and not comparing the current vulnerability to other similar vulnerabilities, specifically those the developer has already fixed. Despite these limitations, some experts successfully employ procedures and existing tools to correctly resolve potential vulnerabilities Even so, those that know how to reason about the most complex vulnerabilities are still susceptible interruption [23] and disorientation [4].

Perhaps vulnerability-specific checklists [24] based on successful strategies could benefit both novice and expert users. The checklists could be structured hierarchically containing high-level tasks, such as "Determine which attacks are feasible", and subsequently more actionable nested subtasks such as "Determine if a SQL injection attack is feasible" or "Determine if an XSS attack is feasible". Checklists allow users to save the state of their interaction with a vulnerability — for example by eliminating attack vectors they have ruled out — diminishing the consequences of an interruption and context loss. Additionally, each task could include links to resources that relate specifically to that task and tools that could help developers complete the task.

6. THREATS TO VALIDITY

The developers we recruited may not be representative of all developers. We conducted an exploratory study with a small number of developers, including some students and participants who had limited familiarity with security vulnerabilities. Developers with more expertise and experience might ask different questions and require different types of tool support.

Further, participants were equipped with FSB, an open source static analysis tool. Developers using other more sophisticated tools might ask slightly different questions. We cannot and do not claim that we have identified a comprehensive list of questions.

One potential threat is that some participants had not worked in the code of interest for some time by the time they participated in the experiment. This may have led them to ask questions that a developer working in their every day coding environment might not ask. Along the same lines, another threat is that we did not have developers who are responsible for the security of their code in real life; we asked participants to pretend they are in charge of security for the code.

Finally, we only examined developers' interactions with four specific vulnerabilities in one Java project. It remains to be seen what questions developers ask about other types of projects.

7. CONCLUSION

This paper reported on an experiment conducted to discover the questions developers ask when identifying and assessing security vulnerabilities in software code. During the experiment, we asked ten software developers to describe their thoughts as they assessed potential security vulnerabilities in iTrust, a medical records web application. We presented the results of our experiment as a catalog of categorized questions. Our findings have several implications for the design of program analysis tools. Our results suggest that tools should help developers, among other things, navigate from malicious data sources to sensitive data sinks within an application and manage the all the tasks required to reason about a vulnerability.

8. REFERENCES

[1] A. Austin and L. Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 97–106. IEEE, 2011.

[2] M. Barnett, R. DeLine, A. Lal, and S. Qadeer. Get me here: Using verification tools to answer developer questions. Technical Report MSR-TR-2014-10, February 2014.

[3] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244. ACM, 2002.

[4] B. de Alwis and G. C. Murphy. Using visual momentum to explain disorientation in the eclipse ide. In *Proceedings of the Visual Languages and Human-Centric Computing*, VLHCC '06, pages 51–54, Washington, DC, USA, 2006. IEEE Computer Society.

[5] L. Dukes, X. Yuan, and F. Akowuah. A case study on web application security testing with tools and manual testing. In *Southeastcon, 2013 Proceedings of IEEE*, pages 1–6. IEEE, 2013.

[6] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 175–184. ACM, 2010.

[7] T. Fritz, G. C. Murphy, E. Murphy-Hill, J. Ou, and E. Hill. Degree-of-knowledge: Modeling a developer's knowledge of code. *ACM Trans. Softw. Eng. Methodol.*, 23(2):14:1–14:42, Apr. 2014.

[8] B. G. Glaser and A. L. Strauss. *The discovery of grounded theory: Strategies for qualitative research*. Transaction Publishers, 2009.

[9] W. Hudson. *Card Sorting*. The Interaction Design Foundation, Aarhus, Denmark, 2013.

[10] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681. IEEE, 2013.

[11] N. Jovanovic, C. Kruegel, and E. rda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.

[12] A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158. ACM, 2004.

[13] O. Kononenko, D. Dietrich, R. Sharma, and R. Holmes. Automatically locating relevant programming help online. In *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, pages 127–134. IEEE, 2012.

[14] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):pp. 159–174, 1977.

[15] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE*

International Conference on Software Engineering-Volume 1, pages 185–194. ACM, 2010.

[16] T. D. LaToza and B. A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, page 8. ACM, 2010.

[17] T. D. LaToza and B. A. Myers. Visualizing call graphs. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 117–124. IEEE, 2011.

[18] S. Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 7(4):325–339, 1987.

[19] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Usenix Security*, pages 18–18, 2005.

[20] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *ACM SIGPLAN Notices*, volume 40, pages 365–383. ACM, 2005.

[21] E. Murphy-Hill, R. Jiresal, and G. C. Murphy. Improving software developers' fluency by recommending development environment commands. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 42:1–42:11, New York, NY, USA, 2012. ACM.

[22] J. Nielsen, T. Clemmensen, and C. Yssing. Getting access to what goes on in people's heads?: reflections on the think-aloud technique. In *Proceedings of the second Nordic conference on Human-computer interaction*, pages 101–110. ACM, 2002.

[23] C. Parnin and S. Rugaber. Programmer information needs after memory failure. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 123–132. IEEE, 2012.

[24] K. Y. Phang, J. S. Foster, M. Hicks, and V. Sazawal. Triaging checklists: a substitute for a phd in static analysis. *Evaluation and Usability of Programming Languages and Tools (PLATEAU) PLATEAU 2009.*

[25] F. Servant and J. A. Jones. History slicing: assisting code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 43. ACM, 2012.

[26] P. Viriyakattiyaporn and G. C. Murphy. Improving program navigation with an active help system. In *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '10, pages 27–41, Riverton, NJ, USA, 2010. IBM Corp.

[27] Y. Yoon, B. A. Myers, and S. Koo. Visualization of fine-grained code change history. In *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*, pages 119–126. IEEE, 2013.